Programmation Fonctionnelle

Fonction de première classe

C# supporte les fonctions de première classes (first-class functions) car il permet de manipuler des fonctions comme des variables, ceci notamment grace aux délégués.

Délégués

Les délégués (delegates) sont le spport de première classe pour les fonctions en C#. Les délégués sont des pointeurs de fonction typés, qui permettent de manipuler les méthodes comme des variables et de les appeler. Le délégué se définit avec le mot-clé `delegate` et contraint la signature de fonction correspondante :

```
public delegate int MathOperation(int leftOperand, int rightOperand);
public class Addition{
public int Add(int leftOperand, int rightOperand) => leftOperand + rightOperand;
}

MathOperation addition = new Addition().Add;
```

On peut ensuite exécuter la méthode à partir de cette variable :

```
var result = addition.Invoke(4,2);
var result = addition(4,2);
```

Lambdas

On peut aussi utiliser la syntaxe lambda pour définir des fonctions anonymes, et les assigner à des délégués :

Lambda "expression body" :

```
MathOperation addition = (int left, int right) => left + right;
```

Lambda normale :

```
MathOperation addition = (int left, int right) => {
  return left + right;
};
```

LinQ

LinQ, pour Language Integrated Queries est une fonctionnalités de C# qui permet d'utiliser les opérateurs fonctionnels sur les collections.

`IEnumerable`

L'interface `IEnumerable` est une abstraction de toute collection sur laquel fonctionne LinQ. Pour générer une `IEnumerable` dans une méthode, on utilise le mot clé `yield` :

```
public static IEnumerable<int> Power(int number, int exponent)
{
  int result = 1;

  for (int i = 0; i < exponent; i++)
  {
    result = result * number;
    yield return result;
  }
}
```

On peut ainsi l'énumérer comme une collection avec une boucle `foreach` :

```
foreach (int i in Power(2, 8))
{
  Console.Write("{0} ", i);
}
```

Opérateurs fonctionels sur collections

LinQ fournit des opérateur fonctionels pour écrire du code très expressif de traitement des collections (toutes les collections de la Base Class Library (BCL) implémentent `IEnumerable`). Le nommage des méthodes rappellent le SQL. LinQ peut s'exécuter sur des données en mémoire ou en dehors du système par le biais des arbres d'expressions. Les méthodes LinQ ont en argument des délégués, on les utilise souvent en passant des lambdas.

```
public class Employee {
    public string Name{get;set;}
    public int Age {get;set;}
}

List<Employee> employees = new List<Employee> {
    new Employee{ Name = "Shepard", Age = 28},
    new Employee{ Name = "Liara", Age = 106}
};

List<string> seniorNames = employees
    .Where(employee => employee.Age >= 50) // Opérateur de filtrage
    .Select(employee => employee.Name) // Opérateur de transformation
    .ToList(); // opérateur terminal

// Resultat : { "Liara" }
```

Les opérateurs de filtrage et de transformation sont "lazy" ; ils ne font rien tant que l'on pas appliqué un opérateur terminal qui va, lui, procéder à l'énumération.

Opérateurs de filtrage

`Where`

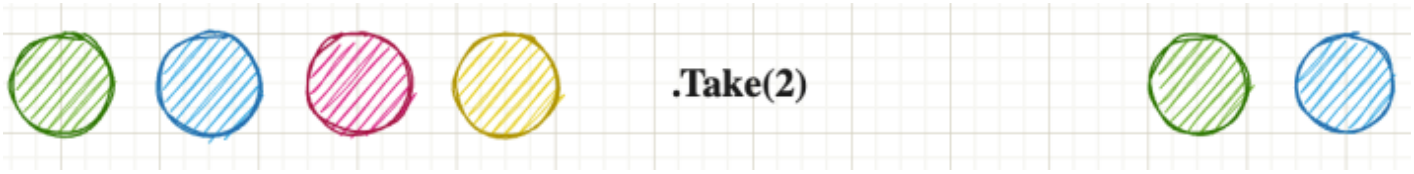
Filtre les éléments à partir d'un prédicat, il conserve les éléments qui valident le prédicat.


```
List<Employee> employees = new List<Employee> {
    new Employee{ Name = "Shepard", Age = 28},
    new Employee{ Name = "Liara", Age = 106}
};

IEnumerable<Employee> seniors = employees
    .Where(user => user.Age >= 18);
```

```
// Resultat : [ Employee {Name = "Liara", Age = 106} ]
```

Take

Prends seulement les n premiers éléments.




```
List<Employee> employees = new List<Employee> {
  new Employee{ Name = "Shepard", Age = 28},
  new Employee{ Name = "Liara", Age = 106},
  new Employee{ Name = "Tali", Age = 23}
};


IEnumerable<Employee> adults = users
.Take(2);

// Resultat : [ Employee {Name = "Shepard", Age = 28}, Employee {Name = "Liara", Age = 106} ]
```

Skip

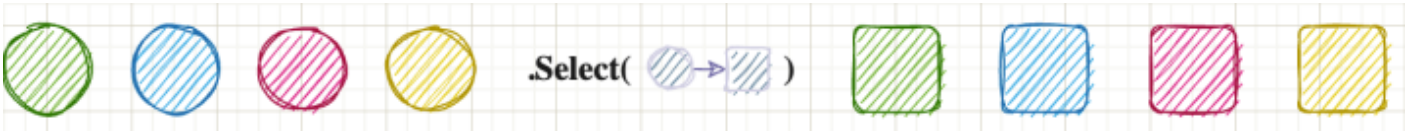
Prends seulement les éléments après en avoir supprimé n.


```
List<Employee> employees = new List<Employee> {
  new Employee{ Name = "Shepard", Age = 28},
  new Employee{ Name = "Liara", Age = 106},
  new Employee{ Name = "Tali", Age = 23}
};

IEnumerable<Employee> adults = users
.Skip(2);

// Resultat : [ Employee {Name = "Tali", Age = 23} ]
```

Opérateurs de transformation

`Select`

Mappe chaque élément à autre chose en utilisant une fonction.




```
List<Employee> employees = new List<Employee> {
  new Employee{ Name = "Shepard", Age = 28},
  new Employee{ Name = "Liara", Age = 106},
  new Employee{ Name = "Tali", Age = 23}
};

IEnumerable<string> employeesNames = employees
  .Select(employee => employee.Name)

// Resultat : [ "Shepard","Liara","Tali" ]
```

`SelectMany`

Mappe chaque élément à une collection d'autre chose, puis applatis le résultat.


```
public class Team {
public string Name {get;set;}
    public List<Employees> Members {get;set;}
}

List<Team> teams = new List<Team> {
  new Team {
      Name = "First Team",
      Members =  new List<Employee> {
          new Employee{ Name = "Shepard", Age = 28},
          new Employee{ Name = "Liara", Age = 106},
          new Employee{ Name = "Tali", Age = 23}
      };
```

```
    },
    new Team {
      Name = "Second Team",
      Members =  new List<Employee> {
          new Employee{ Name = "Garrus", Age = 27},
          new Employee{ Name = "Kaidan", Age = 34},
          new Employee{ Name = "Joker", Age = 30}
      };
    },
}

IEnumerable<string> employeesName = teams
  .SelectMany(team => team.Members) // Retourne IEnumerable<Employee>
  .Select(employee => employee.Name)
```

Opérateurs Terminaux

`ToList` / `ToArray`

Enumère l'expression LinQ pour construire une liste / un tableau.

`First` / `FirstOrDefault`

Enumère l'expression LinQ, jusqu'à trouver un élément qui correspond au prédicat. `First` jette une exception si pas d'élément correspondant, `FirstOrDefault` retourne `null`.


```
List<Employee> employees = new List<Employee> {
  new Employee{ Name = "Shepard", Age = 28},
  new Employee{ Name = "Liara", Age = 106},
    new Employee{ Name = "Tali", Age = 22}
};

Employee tali = employees
.First(user => user.Name == "Tali");
```

Pour aller plus loin

LinQ possède d'autres opérateurs intéressants :

- Aggrégation : `Count`, `Max`, `Aggregate`, `GroupBy` (Terminaux)
- Quantification : `All`, `Any` (Terminaux)
- Fusion : `Join`, `Zip` (Transformation)
- Ensemblistes : `Union`, `Intersect` (Transformation)

Pattern Matching

Le pattern matching consiste à tester une expression, pour vérifier si elle a certaines caractéristiques. On peut l'utiliser en C# dans les `if` (avec le mot-clé `is`) et dans les `switch`-expression.

Vérification de `null`

En utilisant un pattern de type non-nullable :

```
int? maybe = 12;

if (maybe is int number)
{
    Console.WriteLine($"The nullable int 'maybe' has the value {number}");
}
else
{
    Console.WriteLine("The nullable int 'maybe' doesn't hold a value");
}
```

En utilisant le pattern `not` avec `null` :

```
string? message = "This is not the null string";

if (message is not null)
{
    Console.WriteLine(message);
}
```

Vérification de type

```
Animal animal = ...;
if(animal is Human human){
  ...
}
```

Valeur discrètes

```
public State PerformOperation(string command) =>
   command switch
   {
       "SystemTest" => RunDiagnostics(),
       "Start" => StartSystem(),
       "Stop" => StopSystem(),
       "Reset" => ResetToReady(),
       _ => throw new ArgumentException("Invalid string value for command", nameof(command)),
   };
```

Pattern logique

```
public string GetGreeting(string name, TimeOnly time) => time.Hour switch
{
    >=0 and <6 or >=20 => $"{GOOD_NIGHT_MESSAGE} {name}!",
    >=6 and <12 => $"{GOOD_DAY_MESSAGE} {name}!",
    >=12 and <20 => $"{GOOD_AFTERNOON_MESSAGE} {name}!",
    _ => throw new ArgumentException($"Unsupported hour :
{this.currentTimeProvider.CurrentTime.Hour}")
}
```

Pattern de propriétés

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        { Items: > 10, Cost: > 1000.00m } => 0.10m,
        { Items: > 5, Cost: > 500.00m } => 0.05m,
        { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't calculate discount on
null order"),
```

```
        var someObject => 0m,
    };
```

Revision #2
Created 2024-11-21 14:57:25 UTC by Nicolas
Updated 2024-11-21 14:58:28 UTC by Nicolas