Fonctions d'ordre supérieur en C#

Les fonctions d'ordre supérieur sont un concept important en programmation qui permet de passer des fonctions comme des arguments à d'autres fonctions. Cela peut sembler complexe au début, mais avec cet article, vous allez comprendre les bases de ces fonctions et apprendre à les utiliser en C#.

Qu'est-ce qu'une fonction d'ordre supérieur ?

Une fonction d'ordre supérieur est une fonction qui prend une autre fonction comme argument ou retourne une fonction comme valeur de retour. Les fonctions d'ordre supérieur sont également connues sous le nom de fonctions de haut niveau ou de fonctions de deuxième ordre.

Types de fonctions d'ordre supérieur

Une fonction est d'ordre supérieur si elle contient au moins un argument de type `Action` ou `Func`.

1. ACTION (void)

Définit un type de fonction sans valeur de retour (void). Un exemple en C# avec un paramètre `int x` correspond à ceci:

```
void FSuperior(Action<int> x)
{
    x(1);
}
```

Si la fonction en paramètre n'a pas de paramètre:

```
void FSuperior(Action x)
{
    x();
}
```

> On peut en déduire que les paramètres d'entrée sont spécifiés à l'intérieur des chevrons (*généricité*...) du type `Action` :
>
> Action<param1,param2,param3e...>

Exemple d'utilisation de `Action` pour afficher logger messages (*clone* de Console.WriteLine)

```
// Différentes manières de logger les messages

// #1 : méthode "classique", dans un fichier
void MethodToFile(string text)
```

```
{
    File.AppendAllText("log.txt", "(" + DateTime.Now.ToString() + ", Method) " +
text + Environment.NewLine);
}

// #2 : Méthode avec lambda, dans un fichier
Action<string> ActionToFile = text => File.AppendAllText("log.txt", "(" +
DateTime.Now.ToString() + ", Action) " + text + Environment.NewLine);

// #1 : méthode "classique", dans la console
void MethodToConsole(string text)
{
    Console.WriteLine("(" + DateTime.Now.ToString() + ", Method) " + text);
}

// #2 : Méthode avec lambda, dans la console
Action<string> ActionToConsole = text => Console.WriteLine("(" +
DateTime.Now.ToString() + ", Method) " + text);


// Ici, on déclare l'outil de logging. Toute l'application se sert de Log pour
logger
Action<string> Log = MethodToFile;
// Action<string> Log = ActionToFile;
// Action<string> Log = MethodToConsole;
// Action<string> Log = ActionToConsole;

// Et maintenant on logge sans se soucier de où ça va
Log("Using 'void ToFile(string text)'");
```

2. FUNC (Avec valeur de retour)

Si la fonction passée en paramètre retourne une valeur, on utilise le type `Func` et la valeur de retour correspond au dernier type indiqué entre les chevrons (dans l'exemple ci-dessous, `double`):

```
void FSuperior(int a,int b,Func<int, int, double> x)
{
    var z = x(a, b);
    Console.WriteLine(z);
}

//Appel de F
FSuperior(1,2,Add); //3

//Définiton d'une fonction Add respectant les critères du pointeur
Func<int,int,double>
double Add(int a,int b)
{
    return Convert.ToDouble(a + b);
}
```

> Derrière les décors, Action est une sorte de Func<...,void>

Avec valeur de retour pour la fonction de base et son paramètre:

```
int FSuperior1_2(Func<int, int, int> x)
{
    return x(1, 2);
}

//Appel de F
var result = FSuperior1_2(Sub); //-1

//Définiton d'une fonction Sub respectant les critères du pointeur
Func<int,int,int>
int Sub(int a,int b)
{
    return a-b;
}
```

Exemple d'utilisation de `Func` pour doubler une valeur

```
int X2(int x)
{
    return x + x;
}

Func<int, int> X2Alias = X2; //création d'un alias
int result = X2Alias(5); // Retourne 10
```

Exemple d'utilisation de `Func` pour créer une fonction qui retourne une fonction avec un `lambda`

```
Func<int, Func<int, int>> myFunc = x => y => x + y;
int result = myFunc(5)(3); // Retourne 8
```

> Le lambda est expliqué ci-après

Fonction anonyme

Une fonction peut être *déclarée à la volée* sans avoir de nom, on appelle cela un `lambda`. La définition d'un lambda utilise l'opérateur **=>** et on doit comprendre qu'à

- **gauche** de la flèche sont décrits les paramètres (signature)

et qu'à

- **droite** est décrit le comportement (corps).

Signature

Pour spéficier les arguments avec un lambda, il y a 3 cas principaux

1. Pas d'argument

```
//Fonction qui n'a pas de paramètre et retourne la valeur 1
Func<int> one = () => 1;
```

> Ici les parenthèses sont obligatoires

2. Argument non utilisé (underscore)

```
//Fonction avec un paramètre dont le lambda ne tient pas compte et retourne la
valeur 2
Func<int,int> two = _ => 2;
```

3. Argument utilisé

```
//Fonction avec un paramètre et retourne la valeur doublée
Func<int,int> x2 = x => x*2;
```

4. Plusieurs arguments utilisés

```
//Fonction avec 2 paramètres et retourne l'addition
Func<int,int,int> add = (x,y) => x + y;
```

5. Pas tous les arguments utilisés

```
//Fonction avec un paramètre utilisé (x) et un paramètre non utilisé et retourne
la valeur doublée
Func<int,int,int> add2 = (x,_) => x + 2;
```

> À noter que les parenthèses des arguments sont optionnelles (sauf s'il n'y en a pas)

```
//Fonction avec un paramètre et retourne la valeur doublée
Func<int,int> x2 = (x) => x*2;
```

Action et lamdba combinés

La force du lambda est de le combiner avec le type action pour ensuite pouvoir le passer à à une collection pour effectuer une opération sur ses éléments...

Voici donc un exemple de définition d'une `Action` avec un `lambda` (fonction anonyme) :

```
Action<int> Print = x => Console.WriteLine(x);
Print(5); // Affichera "5" dans la console
```

Dans cet exemple, `Print` est une fonction qui prend un entier comme argument et affiche ce nombre dans la console.

Si la fontion est complexe, le corps d'un lambda peut être écrit comme pour une fonction avec des `accolades` (sans oublier le point virgule à la fin):

```
Action<int> Print = x =>
{
    //Corps de fonction standard
    Console.WriteLine(x);
};
```

Fonction nommée

On peut faire la même chose avec une fonction *standard* (pas anonyme):

```
//Code de la classe
Action<int> PrintThisAlias = PrintThis;

//Fonction définie dans une classe
void PrintThis(int x)
{
    Console.WriteLine(x);
}
```

Autres Exemples

Exemple A

Voici un exemple d'utilisation de `Action` avec LINQ pour filtrer une collection :

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
Action<int> PrintOdd = x => {
    if (x % 2 == 1)
    {
        Console.WriteLine(x);
    }
```

```
};
numbers.ForEach(PrintOdd); // 1,3,5
```

Dans cet exemple, `PrintOdd` est une fonction qui prend un entier comme argument et affiche ce nombre dans la console. La méthode `ForEach` de LINQ utilise cette fonction pour appeler la fonction sur chaque élément **sans écrire de boucle for**.

Exemple B

Voici un exemple d'utilisation de `Func` avec LINQ pour filtrer une collection :

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
Func<int, bool> isEven = x => x % 2 == 0;
numbers.Where(isEven).ToList(); // Retourne la liste des nombres pairs
```

Dans cet exemple, `isEven` est une fonction qui prend un entier comme argument et retourne un booléen indiquant si le nombre est pair. La méthode `Where` de LINQ utilise cette fonction pour filtrer la collection `numbers` et retourner la liste des nombres pairs, qui elle-même est convertie en liste.

Avantages des fonctions d'ordre supérieur

Les fonctions d'ordre supérieur offrent plusieurs avantages, notamment :

1. Flexibilité

Les fonctions d'ordre supérieur permettent de créer des fonctions plus génériques et plus flexibles. Vous pouvez passer des fonctions comme arguments pour les modifier ou les combiner.

2. Réutilisation

Les fonctions d'ordre supérieur permettent de réutiliser du code en passant des fonctions comme arguments. Vous pouvez créer des fonctions qui peuvent être utilisées avec différentes fonctions.

3. Simplification

Les fonctions d'ordre supérieur peuvent simplifier le code en permettant de regrouper des opérations complexes en une seule fonction.

Exemples d'utilisation avec LINQ et lambda

1. Utilisation de `Action` pour afficher des messages ()

```
List<string> names = new List<string> { "John", "Mary", "Jane" };
Action<string> Print = name => Console.WriteLine(name);
names.ForEach(Print); // Affichera les noms dans la console
```

2. Utilisation de `Func` pour filtrer une collection

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
Func<int, bool> IsEven = x => x % 2 == 0;
Func<int, bool> IsBig = x => x  > 2;
numbers.Where(IsEven).ToList(); // Retourne la liste des nombres pairs (2,4)
numbers.Where(IsBig).ToList();  // Retourne la liste des "grands" nombre (3,4,5)
```

On relèvera le fait que la méthode qui effectue l'action de filtrage est la même dans les deux cas: `Where`

Conclusion

Action et **Func** sont donc 2 nouveaux *types* à connaître et maîtriser en C# afin de tirer le maximum de la programmation fonctionnelle et de comprendre ses mécanismes sous-jacents.

La notion de fonction anonyme nomée `lambda` avec son opérateur `=>` est aussi un élément clé de l'integration de la programmation fonctionnelle avec C#.

De manière générale, les fonctions d'ordre supérieur sont un concept important en programmation qui peut sembler complexe au début, mais qui offre de nombreux avantages. En comprenant les fonctions d'ordre supérieur, vous pouvez créer des programmes plus flexibles, plus réutilisables et plus simples.

Source

Inspiré de groq ai inference avec le prompt suivant :

```
théorie sur les fonctions d'ordre supérieur pour des débutants en programmation
Csharp en markdown et avec maximum 500 mots
```