BOUNATIROU Rodolphe

Centre de readaptation de mulhouse  - 2023

Fonction flechees en c#

# Introduction

**Le code est valide sous la version 6 de .NET**

Les fonctions fléchées, appelée autrement fonction anonyme, expressions lambda ou encore fonction lambda bien que les deux dernières aient une petite différence quant à leur mode de fonctionnement.

Elles sont constitué de paramètres et d’un corps.  
  
Ces deux parties sont séparés par un opérateur lambda noté ( => )

Dans ce cours vous apprendrez plusieurs usages de ces opérateur qui permettront de gagner en productivité.

# Usage pour accélérer l’écriture d’une fonction

Comme dis en introduction une fonction fléchée est constitué de deux parties.  
à gauche les paramètres, à droite un corps de fonction ou une expression.

|  |  |
| --- | --- |
| Fonction Lambda : paramètre1 => { CORPS ; } | Expression Lambda : (paramètre1, paramètre2) =>EXPRESSION |

Notons que :

**-les parenthèse autour des paramètres ne sont pas nécessaire, lorsqu’il y’a 1 seul paramètre,  
si il n’y en a aucun on laisse une parenthèse vide**

- **sans accolades {} autour de la partie à droite de l’opérateur lambda, on appelle cela une expression lambda .**

**- avec accolades {} autour de la partie à droite de l’opérateur lambda, on appelle cela une fonction lambda .**

**- les expressions lambdas impliquent un return de l’expression qui n’a pas besoin d’être formulé.  
- les fonctions lambdas doivent quant à elle utiliser le mot clé return pour leur retour.**

Exemple pratique de Pierre:

Imaginons une classe contenant une fonction ToString() dans une classe Personne

La classe personne ayant des attributs prenom, nom et age  
La fonction renverrait :

* Le nom en uppercase
* Le prenom en lowercase suivi d’une virgule
* L’age suivi du mot ans, au singulier,si il a moins de 2 ans

Exemple pour une instance de personne ayant pour etat

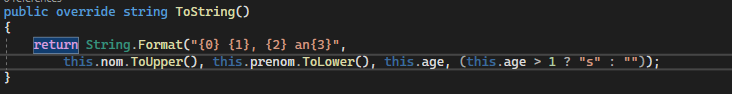
|  |
| --- |
| pierre : Personne |
| nom = «  Anonyme »  prenom = « Pierre »  age = 25 |

On obtiendrait en retour du comportement ToString() de l’instance

« ANONYME pierre, 25 ans »

La fonction pourrait être ainsi codé

On peut également l’écrire en **expression lambda** via l’opérateur lambda



Dans la mesure, où la partie à droite de l’expression lambda est du même type que la méthode,  
l’absence d’accolades autour de la partie gauche indique qu’il s’agit d’une expression lambda.  
S’il y avait eu des accolades, cela aurait été une **fonction lambda :**



**Toutefois, dans ce cas de figure, on ne peut pas se servir des fonctions lambdas**

Donc, si une fonction se limite à un return, alors une expression lambda peut être utilisée. La partie de droite d’une **expression lambda** sert de retour.

# Action et Func

**Action** et **Func** sont des classes dites générique, c’est-à-dire qu’elles contiennent des types de variables non définis que l’on précisera entre les signes < et >.

Une classe générique que vous avez déjà vu est la classe **List**,

Exemple : **List<T>** ou **T est le type non défini**

Entre ses signes <> nous précisons le type de donnée que nous souhaitons stocker

Par exemple, **List<int>** ne stockera que des données compatibles avec le type **int**.

Dans le cas de **Action**, ces types correspondront aux types des arguments à passer en paramètre.

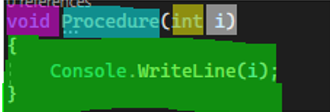
Dans le cas de **Func**, ces types correspondront eux aussi aux types des arguments à passer en paramètre, à l’exception, du dernier des élément qui correspondra au type de retour de la fonction.  
  
La classe **Action** peut se substituer à une **procédure**,  
tandis que la classe **Func** se substituera à une **fonction**.  
Par conséquent, **Action peut être vide** (devenant Action sans être générique) mais **Func devra toujours avoir au moins un type correspondant au type de retour.**

|  |  |
| --- | --- |
| **Méthode** | **Peut-être substitué par un objet du type** |
| int MaFonction(object o, bool p) ; | Func<object,bool,int> MaFonction |
| void MaFonction(object o, bool p); | Action<object,bool> MaFonction |
| **Et dans les deux cas l’appel peut se faire ainsi (dans la mesure ou la portée le permet):** | |
| MaFonction(o,p) ; | |

|  |  |
| --- | --- |
| ⚠ | *Attention :*  *Il y’a une limite de 16 arguments (donc 17 types pour* ***Func*** *et 16 pour* ***Action****)*  *Ce qui laisse quand même beaucoup de marge* |

## Comment les utiliser ?

Supposons une procédure, prenant en paramètre i de type entier (int).



La procédure se contente d’afficher la valeur de i dans la console

Adaptons cette Procédure !

Tout d’abord, il s’agit d’une procédure donc elle retourne void. Cela correspond donc à une **Action.**  
Elle dispose également d’un paramètre int.

Donc son type sera : **Action<int>**.



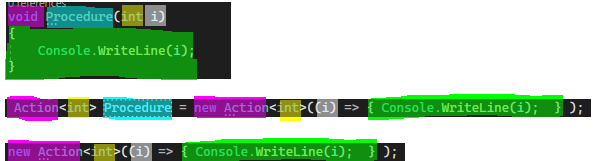
**Le nommage des variables dans les paramètres d’instanciation d’ Action<int> se fait à** gauche de **l’opérateur lambda (=>). S’il n’y a qu’un paramètre, la parenthèse autour de ces derniers est facultative.**

*ici la parenthèse autour du nommage des paramètre à été enlevé*

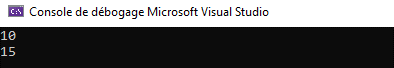
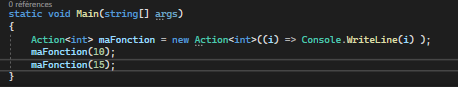


**Et le corps de la méthode,** ici surligné en vert, dans l’image ci-dessus, se retrouve **à droite de l’opérateur.**

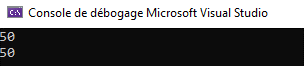
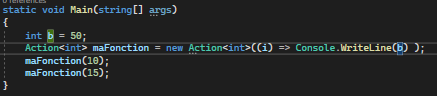
*Notons que lorsque l’on utilise une expression lambda, en général ce sera lié à un type Func car l’expression lambda fait un return de manière tacite.*



Pour son utilisation on peut utiliser le nom qui est donné à l’instance afin de l’utiliser de la même manière qu’une fonction

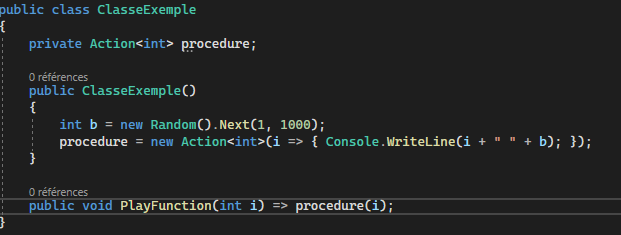
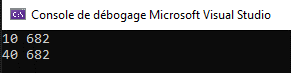
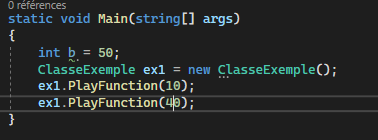


Comme l’action est créé dans un bloc d’instruction, l’un des avantages est que l’on peut utiliser des variables locales au bloc dans lequel l’instance de l’**Action**, ou de la **Func**, est contenu comme arguments pouvant être utilisé par cette dernière.

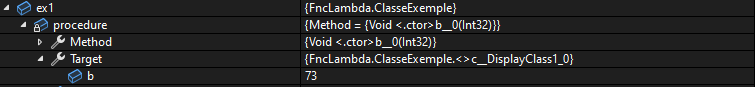


Ainsi la variable b du bloc Main peut être utilisée.

Un autre exemple



La variable b à été conservée (bien que locale au constructeur) dans une propriété Target de l’instance Action (nommée procedure). *(voir ci-dessous)*  
Grâce à cela , les variables nécessaires à l’Action sont conservés et réutilisables dans le corps de cette dernière. (ici ce n’est pas 682 mais 73 car il s’agit d’une autre relance du programme)



Petite info :  
Le framework dotnet permet dans certain cas de se passer du new **Action** ou du new **Func**.  
Cela en convertissant automatiquement l’expression lambda dans le type correspondant.

# Mise en pratique

## Contexte

Imaginons que nous souhaitions développer un logiciel,  
qui à l’aide d’une IHM nous permettra de gérer les feux de circulation d’une ville.  
Dans cette IHM, nous aurions un visuel de l’état de chacun des feux.

Les feux de circulations passent de manière cyclique dans 3 état vert, orange et rouge.  
  
L’IHM nous permettrait de forcer le passage du feu à un état. Mais seulement si le feu n’est pas déjà dans l’état donné. Auquel cas le menu (ToolStripItem par exemple) de changement d’état serait désactivé le temps que le feux ne soit plus dans le même état que l’état désiré.

## Ressources

Pour les ressources nécessaires, veuillez-vous appuyez sur l’exemple du dépôt git ci-dessous en le clonant :

|  |  |
| --- | --- |
| **CMD** | git clone https://github.com/rbounatirou/ExempleFncFleches.git |

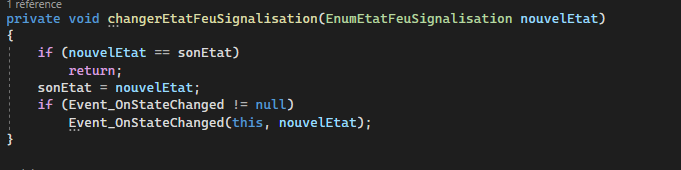
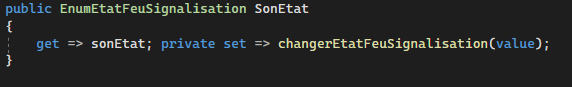
## Analyse

### Classe métier

Décortiquons la classe métier dans le projet FeuxSignalisations  
Elle possède un delegate auquel est rattaché un event   
  
Le mutateur[[1]](#footnote-1) privé « SonEtat » (public en accesseur) va permettre en plus de changer l’état du FeuSignalisation d’informer les personnes rattaché à l’event du changement d’état



C’est au travers du mutateur que les changements d’état se feront, afin de mettre à jour l’état, et de générer l’évènement pour les abonnés à ce dernier.



*L’état de la signalisation sera régulièrement changé dans un thread lancé dans le constructeur.*

### Coté IHM

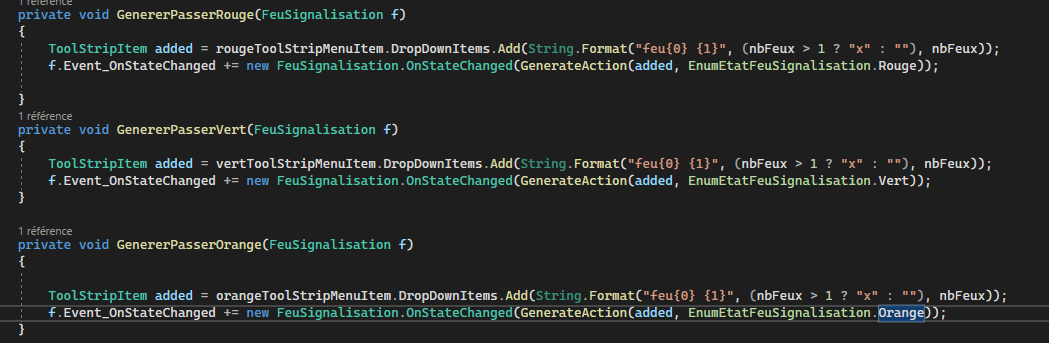
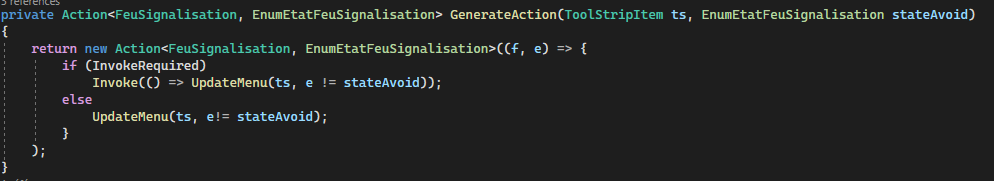
Les ToolStripItem sont générés dynamiquement lors de la création de feux.  
(et un UserControl est ajouté)  
  
Les ToolStripItem ainsi généré sont rattaché à l’évènement Event\_OnStateChanged de la classe métier.

La propriété enable du ToolStripItem change en fonction de l’état du feu qui lui est « lié ».

En théorie pour faire cela sans fonction fléché on pourrait mettre à chaque FeuSignalisation un attribut unique permettant de l’identifier, et rattacher via la propriété Tag du menu ToolStripItem le FeuSignalisation correspondant.  
  
Ensuite en abonnant l’IHM à l’event de chacun des feu,   
lors du changement d’état d’un feu (sender), on regarderait l’ensemble des ToolStripItem pour actualiser ceux dont le Tag correspond au Sender.

Bref ce système force beaucoup de boucle pour les recherches, la création d’un identifiant unique dans la classe métier,  
et par certain aspect il complexifie la programmation des composants.  
  
Afin de limiter toutes ces adaptations on va utiliser les fonctions fléchées

Grace aux fonctions fléchées, on générera des Action pour remplir le rôle de procédure abonnée à l’événement. (remplissant la signature du délégué)



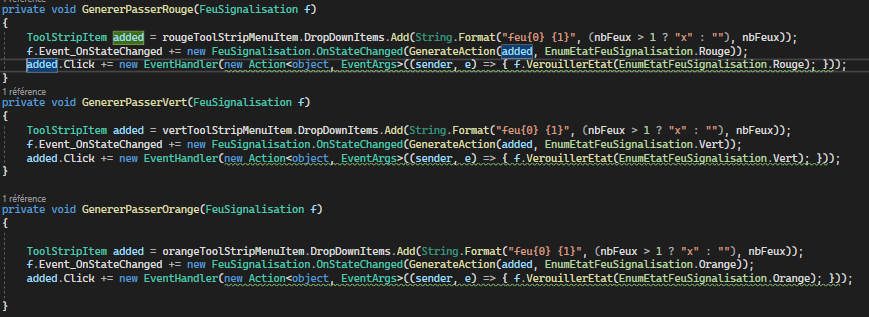
Comme la procédure est générée (et ainsi propre à chaque instance), elle a accès à d’autres paramètres que ceux qu’impose le délégué. **C’est un des grands intérêts des fonctions fléchées, il permet d’avoir accès à des données normalement inaccessible si on passait par une méthode associée à l’évènement déléguée. (En les passant en Targets)**

**Les attributs de la propriété target(voir page 7), conservent les variable (ci dessuss notre ToolSripItem généré dynamiquement *added*) et permet d’y avoir accès sans passer par des paramètres de la méthode.  
Ainsi la signature du délégué peut être respecté tout en ajoutant les données nécessaires.**  
  
Le code est donc simplifiée d’une certaine façon (même si cela demande un peu d’apprentissage supplémentaire). De plus dans notre cas de figure il va limiter le parcours pour retrouver le Tag… etc.

**Pour comparer la facilité de désactivation des menus, une version sans fonctions fléchées est disponible dans le projet.**

**Par ailleurs je vous invite à ajouter à chaque ToolStripItem un événement Click qui utilisera la méthode VerouillerEtat de la classe métier de l’instance correspondante.**

### Solution



# Autre cas d’utilisation

En dehors de Action et Func on peut générer des fonctions ou expression fléchées pour remplir de  
nombreux rôles.

## Comparison<T>

Pour les comparaison le délégué Comparison<T> peut être utilisé.  
le délégué Comparison<T> est utile dans la méthode Sort de la classe List<T>

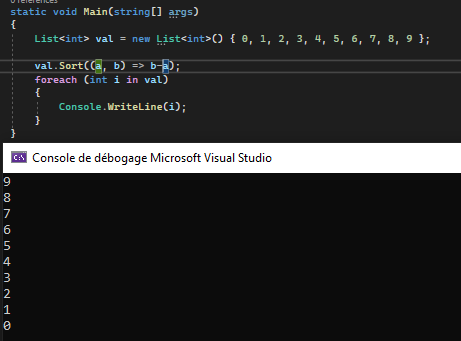
Sa signature est :

public delegate int Comparison<in T>(T x, T y);

elle renvoie un int qui correspond à l’ordre

|  |  |
| --- | --- |
| Cas | Valeur retournée |
| x<y | inferieur à 0 |
| x==y | 0 |
| x>y | supérieur à 0 |

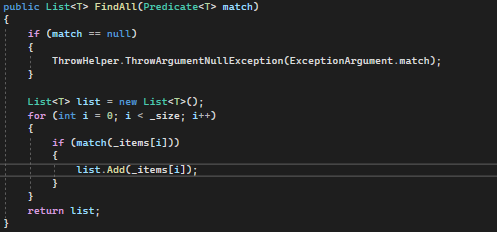
On pourrait donc lui donnée une expression lambda qui trie par ordre décroissant en   
faisant



## Predicate<T>

Pour savoir si quelque chose est vrai ou faux il existe un délégué : le prédicat nommé Predicate.

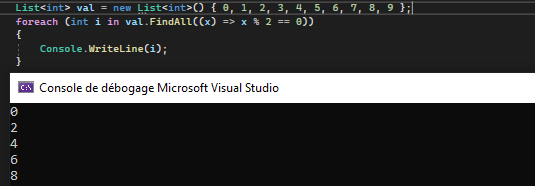
public delegate bool Predicate<in T>(T obj);



Predicate va permettre dans certaines fonctions comme **Find** et **FindAll**  de **List<T>** d’établir à l’aide d’une expression booléenne des critères de sélection.

D’ailleurs vous pouvez voir ci-dessus la manière dont est codée la fonction **FindAll** de la classe **List<T>**

Exemple ne trouver que les nombres pairs.



## Converter<TInput, TOutput>

Le Converter va permettre des conversions de type.

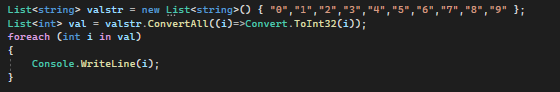
Son délégué à pour signature.

**public delegate TOutput** Converter<**in** TInput,out TOutput>(TInput input);

**La partie out du Converter n’est pas à préciser dans les paramètres de la fonction fléchée mais est définie automatiquement selon le type de retour.**

Le code ci-dessous va permettre de convertir la List<string> en List<int> de manière simple

(pour cet exemple j’ai réécris la deuxième ligne de manière plus complète en dessous pour une  
compréhension plus claire tout en montrant le côté facultatif de cette précision)



La classe List<T> possède beaucoup de méthodes utilisant les délégués par conséquent elle  
est un bon moyen d’entrainement à l’utilisation des fonction fléchées.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Nom méthode** | **Action** | **Func** | **Predicate** | **Comparison** | **Converter** |
| ***Exists*** |  |  | X |  |  |
| ***ConvertAll*** |  |  |  |  | X |
| ***Find*** |  |  | X |  |  |
| ***FindAll*** |  |  | X |  |  |
| ***FindIndex*** |  |  | X |  |  |
| ***FindLast*** |  |  | X |  |  |
| ***ForEach*** | X |  |  |  |  |
| ***RemoveAll*** |  |  | X |  |  |
| ***Sort*** |  |  |  | X |  |
| ***TrueForAll*** |  |  | X |  |  |

Pour vous entrainer, des exercices sont à votre disposition dans une solution clonée avec la mise en pratique de ce document.

1. Mutateur = (Syn.Setter) Sert à la modification d’un attribut de l’instance. (Parfois appelé modifieur) [↑](#footnote-ref-1)