

Expressions lambda (Guide de programmation C#)

29/07/2019 • 9 minutes de lecture • 

Dans cet article

[Expressions lambdas](#)

[Instructions lambda](#)

[Lambdas asynchrones](#)

[Expressions lambda et tuples](#)

[Lambdas avec les opérateurs de requête standard](#)

[Inférence de type et expressions lambda](#)


[Capture des variables externes et de l'étendue variable dans les expressions lambda](#)
[spécification du langage C#](#)

[Chapitre proposé](#)


[Voir aussi](#)

Une *expression lambda* est une expression de l'une des deux formes suivantes :

- [Expression lambda](#) qui a une expression comme corps :

C#	
<pre>(input-parameters) => expression</pre>	



- [Instruction lambda](#) qui a un bloc d'instructions comme corps :

C#	
<pre>(input-parameters) => { <sequence-of-statements> }</pre>	



Utilisez l'[opérateur de déclaration lambda=>](#) pour séparer la liste des paramètres de l'expression lambda de son corps. Pour créer une expression lambda, vous spécifiez des paramètres d'entrée (le cas échéant) à gauche de l'opérateur lambda et une expression ou un bloc d'instructions de l'autre côté.

Toute expression lambda peut être convertie en type [délégué](#). Le type délégué vers lequel une expression lambda peut être convertie est défini par les types de ses paramètres et de sa valeur de retour. Si une expression lambda ne retourne pas de valeur, elle peut être convertie en l'un des types délégués `Action` ; sinon, elle peut être



convertie en l'un des types délégués `Func`. Par exemple, une expression lambda qui a deux paramètres et qui ne retourne aucune valeur peut être convertie en un délégué [Action<T1,T2>](#). Une expression lambda qui a un paramètre et qui retourne une valeur peut être convertie en un délégué [Func<T,TResult>](#). Dans l'exemple suivant, l'expression lambda `x => x * x`, qui spécifie un paramètre nommé `x` et retourne la valeur du carré de `x`, est assignée à une variable d'un type délégué :

C#		
<pre>Func<int, int> square = x => x * x; Console.WriteLine(square(5)); // Output: // 25</pre>		

Les expressions lambda peuvent également être converties en types d'[arborescence d'expression](#), comme le montre l'exemple suivant :

C#		
<pre>System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x; Console.WriteLine(e); // Output: // x => (x * x)</pre>		

Vous pouvez utiliser des expressions lambda dans tout code qui requiert des instances de types délégués ou d'arborescences d'expressions, par exemple en tant qu'argument de la méthode [Task.Run\(Action\)](#), pour passer le code qui doit être exécuté en arrière-plan. Vous pouvez également utiliser des expressions lambda lorsque vous écrivez [LINQ C#dans](#), comme le montre l'exemple suivant :


C#		
<pre>int[] numbers = { 2, 3, 4, 5 }; var squaredNumbers = numbers.Select(x => x * x); Console.WriteLine(string.Join(" ", squaredNumbers)); // Output: // 4 9 16 25</pre>		

Lorsque la méthode [Enumerable.Select](#) est appelée avec une syntaxe fondée sur une méthode dans la classe [System.Linq.Enumerable](#) (par exemple, dans LINQ to Objects et LINQ to XML), le paramètre est un type délégué [System.Func<T,TResult>](#). Si la méthode [Queryable.Select](#) est appelée dans la classe [System.Linq.Queryable](#) (par exemple, dans LINQ to SQL), le paramètre est un type d'arborescence d'expression [Expression<Func<TSource,TResult>>](#). Dans les deux cas, vous pouvez utiliser la même expression lambda pour spécifier la valeur de paramètre. Les deux appels `select`

semblent ainsi semblables alors qu'en fait le type des objets créés à partir des lambdas est différent.


Expressions lambdas

Une expression lambda comportant une expression à droite de l'opérateur `=>` est appelée *expression lambda*. Les expressions lambda sont utilisées en grand nombre dans la construction d'[arborescences d'expressions](#). Une expression lambda retourne le résultat de l'expression et prend la forme de base suivante :


C#	
<pre>(input-parameters) => expression</pre>	

Les parenthèses sont facultatives uniquement si le lambda comporte un paramètre d'entrée ; sinon, elles sont obligatoires.


Spécifiez des paramètres d'entrée de zéro avec des parenthèses vides :

C#	
<pre>Action line = () => Console.WriteLine();</pre>	

Les paramètres d'entrée sont séparés par des virgules entre parenthèses :

C#	
<pre>Func<int, int, bool> testForEquality = (x, y) => x == y;</pre>	

Il est parfois impossible pour le compilateur de déduire les types d'entrée. Vous pouvez spécifier les types explicitement comme dans l'exemple suivant :

C#	
<pre>Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;</pre>	


Les types de paramètres d'entrée doivent être tous explicites ou tous implicites ; sinon, une erreur de compilateur [CS0748](#) se produit.

Le corps d'une expression lambda peut se composer d'un appel de méthode. Cependant, si vous créez des arborescences d'expressions évaluées en dehors du contexte du common language runtime .NET, comme dans SQL Server, vous ne devez

pas utiliser d'appels de méthode dans les expressions lambda. Les méthodes n'auront aucune signification en dehors du contexte du Common Language Runtime .NET.

Instructions lambda

Une instruction lambda ressemble à une expression lambda, mais l'instruction ou les instructions sont mises entre accolades :

C#	
<pre>(input-parameters) => { <sequence-of-statements> }</pre>	


Le corps d'une instruction lambda peut se composer d'un nombre illimité d'instructions ; toutefois, en pratique, leur nombre est généralement de deux ou trois.

C#		
<pre>Action<string> greet = name => { string greeting = \$"Hello {name}!"; Console.WriteLine(greeting); }; greet("World"); // Output: // Hello World!</pre>		

Les instructions lambda ne peuvent pas être utilisées pour créer des arborescences d'expression.

Lambdas asynchrones

Vous pouvez facilement créer des expressions et des instructions lambda qui incorporent un traitement asynchrone en utilisant les mots clés [async](#) et [await](#). Par exemple, l'exemple Windows Forms suivant contient un gestionnaire d'événements qui appelle et attend une méthode async `ExampleMethodAsync`.

C#	
<pre>public partial class Form1 : Form { public Form1() { InitializeComponent(); button1.Click += button1_Click; } }</pre>	

```
private async void button1_Click(object sender, EventArgs e)
{
    await ExampleMethodAsync();
    textBox1.Text += "\r\nControl returned to Click event handler.\n";
}

private async Task ExampleMethodAsync()
{
    // The following line simulates a task-returning asynchronous
    process.
    await Task.Delay(1000);
}
}
```

Vous pouvez ajouter le même gestionnaire d'événements en utilisant une expression lambda asynchrone. Pour ajouter ce gestionnaire, ajoutez un modificateur `async` avant la liste des paramètres lambda, comme dans l'exemple suivant :

C#

 Copier

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event
handler.\n";
        };
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous
        process.
        await Task.Delay(1000);
    }
}
```



Pour plus d'informations sur la création et l'utilisation des méthodes asynchrones, consultez [Programmation asynchrone avec `async` et `await`](#).

Expressions lambda et tuples



À compter de sa version 7.0, le langage C# offre une prise en charge intégrée des [tuples](#). Vous pouvez fournir un tuple comme argument à une expression lambda, et

vosre expression lambda peut aussi retourner un tuple. Dans certains cas, le compilateur C# utilise l'inférence de type pour déterminer les types des composants du tuple.

Vous définissez un tuple en plaçant entre des parenthèses une liste de ses composants avec des virgules comme séparateur. L'exemple suivant utilise un tuple à trois composants pour passer une séquence de nombres à une expression lambda, qui double chaque valeur et retourne un tuple à trois composants contenant le résultat des multiplications.

C#		
<pre>Func<(int, int, int), (int, int, int)> doubleThem = ns => (2 * ns.Item1, 2 * ns.Item2, 2 * ns.Item3); var numbers = (2, 3, 4); var doubledNumbers = doubleThem(numbers); Console.WriteLine(\$"The set {numbers} doubled: {doubledNumbers}"); // Output: // The set (2, 3, 4) doubled: (4, 6, 8)</pre>		

En règle générale, les champs d'un tuple sont nommés `Item1`, `Item2`, etc. Toutefois, vous pouvez définir un tuple avec des composants nommés, comme le montre l'exemple suivant.

C#		
<pre>Func<(int n1, int n2, int n3), (int, int, int)> doubleThem = ns => (2 * ns.n1, 2 * ns.n2, 2 * ns.n3); var numbers = (2, 3, 4); var doubledNumbers = doubleThem(numbers); Console.WriteLine(\$"The set {numbers} doubled: {doubledNumbers}");</pre>		

Pour plus d'informations sur les tuples C#, voir [Types tuples C#](#).

Lambdas avec les opérateurs de requête standard

LINQ to Objects, parmi d'autres implémentations, a un paramètre d'entrée dont le type fait partie de la famille [Func<TResult>](#) de délégués génériques. Ces délégués utilisent des paramètres de type pour définir le nombre et le type des paramètres d'entrée, ainsi que le type de retour du délégué. Les délégués `Func` sont très utiles pour l'encapsulation des expressions définies par l'utilisateur appliquées à chaque élément dans un jeu de données sources. Prenons par exemple le type délégué [Func<T,TResult>](#) :

C#

 Copier

```
public delegate TResult Func<in T, out TResult>(T arg)
```

Ce délégué peut être instancié comme une instance `Func<int, bool>`, où `int` est un paramètre d'entrée et `bool` la valeur de retour. La valeur de retour est toujours spécifiée dans le dernier paramètre de type. Par exemple, `Func<int, string, bool>` définit un délégué avec deux paramètres d'entrée, `int` et `string`, et un type de retour `bool`. Le délégué `Func` suivant, lorsqu'il est appelé, retourne une valeur booléenne indiquant si le paramètre d'entrée est égal à cinq :

C#

 Copier Exécuter

```
Func<int, bool> equalsFive = x => x == 5;  
bool result = equalsFive(4);  
Console.WriteLine(result); // False
```

Vous pouvez aussi fournir une expression lambda quand le type d'argument est [Expression<TDelegate>](#), par exemple, dans les opérateurs de requête standard définis dans le type [Queryable](#). Quand vous spécifiez un argument [Expression<TDelegate>](#), l'expression lambda est compilée en arborescence de l'expression.

L'exemple suivant utilise l'opérateur de requête standard [Count](#) :

C#

 Copier Exécuter

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };  
int oddNumbers = numbers.Count(n => n % 2 == 1);  
Console.WriteLine($"There are {oddNumbers} odd numbers in {string.Join(" ",  
numbers)}");
```

Le compilateur peut déduire le type du paramètre d'entrée, ou vous pouvez également le spécifier explicitement. Cette expression lambda particulière compte les entiers (`n`) qui, lorsqu'ils sont divisés par deux, ont un reste de 1.

L'exemple suivant produit une séquence qui contient tous les éléments du tableau `numbers` précédant le 9, car c'est le premier nombre de la séquence qui ne remplit pas la condition :

C#

 Copier Exécuter

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };  
var firstNumbersLessThanSix = numbers.TakeWhile(n => n < 6);  
Console.WriteLine(string.Join(" ", firstNumbersLessThanSix));
```

```
// Output:  
// 5 4 1 3
```

L'exemple suivant spécifie plusieurs paramètres d'entrée en les plaçant entre parenthèses. La méthode retourne tous les éléments du tableau `numbers` jusqu'à ce qu'elle rencontre un nombre dont la valeur est inférieure à sa position ordinaire dans le tableau :

C#

 Copier Exécuter

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };  
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);  
Console.WriteLine(string.Join(" ", firstSmallNumbers));  
// Output:  
// 5 4
```

Inférence de type et expressions lambda

Il n'est généralement pas nécessaire, avec les expressions lambda, de spécifier un type pour les paramètres d'entrée, car le compilateur peut le déduire du corps de l'expression, des types de paramètres et d'autres facteurs, comme le décrit la spécification du langage C#. Pour la plupart des opérateurs de requête standard, la première entrée est le type des éléments dans la séquence source. Si vous interrogez un `IEnumerable<Customer>`, le type de la variable d'entrée est inféré comme étant un objet `Customer`, ce qui signifie que vous avez accès à ses méthodes et à ses propriétés :

C#

 Copier

```
customers.Where(c => c.City == "London");
```

Voici les règles générales de l'inférence de type pour les expressions lambda :

- Le lambda doit contenir le même nombre de paramètres que le type délégué.
- Chaque paramètre d'entrée dans le lambda doit être implicitement convertible en son paramètre de délégué correspondant.
- La valeur de retour du lambda (le cas échéant) doit être implicitement convertible en type de retour du délégué.

Notez que les expressions lambda n'ont pas de type en elles-mêmes, car le système de type commun (CTS, Common Type System) ne comporte aucun concept intrinsèque « d'expression lambda ». Toutefois, il est parfois commode de parler de manière

informelle du « type » d'une expression lambda. Dans ce cas, le type fait référence au type délégué ou au type [Expression](#) dans lequel est convertie l'expression lambda.

Capture des variables externes et de l'étendue variable dans les expressions lambda

Les expressions lambda peuvent faire référence à des *variables externes*. Il s'agit de variables dans l'étendue de la méthode qui définit l'expression lambda, ou dans l'étendue du type qui contient l'expression lambda. Les variables capturées de cette manière sont stockées pour une utilisation dans l'expression lambda, même si les variables se trouvent en dehors de la portée et sont récupérées par le garbage collector. Une variable externe doit être assignée de manière précise pour pouvoir être utilisée dans une expression lambda. L'exemple suivant illustre ces règles :

C#

 Copier

```
public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int> updateCapturedLocalVariable;
        internal Func<int, bool> isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"{j} is greater than {input}: {result}");
            };

            isEqualToCapturedLocalVariable = x => x == j;

            Console.WriteLine($"Local variable before lambda invocation:
{j}");

            updateCapturedLocalVariable(10);
            Console.WriteLine($"Local variable after lambda invocation:
{j}");
        }
    }

    public static void Main()
    {
        var game = new VariableCaptureGame();

        int gameInput = 5;
```

```
game.Run(gameInput);

int jTry = 10;
bool result = game.IsEqualToCapturedLocalVariable(jTry);
Console.WriteLine($"Captured local variable is equal to {jTry}: {result}");

int anotherJ = 3;
game.UpdateCapturedLocalVariable(anotherJ);

bool equalToAnother = game.IsEqualToCapturedLocalVariable(anotherJ);
Console.WriteLine($"Another lambda observes a new value of captured variable: {equalToAnother}");
}
// Output:
// Local variable before lambda invocation: 0
// 10 is greater than 5: True
// Local variable after lambda invocation: 10
// Captured local variable is equal to 10: True
// 3 is greater than 5: False
// Another lambda observes a new value of captured variable: True
}
```

Les règles suivantes s'appliquent à la portée des variables dans les expressions lambda :

- Une variable qui est capturée ne sera pas récupérée par le garbage collector avant que le délégué qui le référence soit disponible pour le garbage collection.
- Les variables introduites dans une expression lambda ne sont pas visibles dans la méthode englobante.
- Une expression lambda ne peut pas capturer directement un paramètre [in](#), [ref](#) ou [out](#) dans la méthode englobante.
- Une instruction [return](#) dans une expression lambda ne provoque pas le retour de la méthode englobante.
- Une expression lambda ne peut pas contenir d'instruction [goto](#), [break](#) ou [continue](#) si la cible de cette instruction de saut se trouve en dehors du bloc de l'expression lambda. Il est également incorrect d'avoir une instruction de saut en dehors du bloc de l'expression lambda si la cible se trouve à l'intérieur du bloc.

spécification du langage C#

Pour plus d'informations, consultez la section [Expressions de fonction anonyme](#) de la [spécification du langage C#](#).

Chapitre proposé

[Delegates, Events, and Lambda Expressions](#) (Délégués, événements et expressions lambda) dans [C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers](#)

Voir aussi

- [Guide de programmation C#](#)
- [LINQ \(Language Integrated Query\)](#)
- [Arborescences d'expression](#)
- [Comparaison entre les fonctions locales et les expressions lambda](#)
- [Expressions lambda implicitement typées](#)
- [Exemples Visual Studio 2008 C# \(voir les fichiers d'exemples de requêtes LINQ et le programme XQuery\)](#)
- [Expressions lambda récursives](#)

Cette page est-elle utile ?

 Oui  Non
