

Les méthodes anonymes de Dotnet 2

par Jérôme Lambert ([Mes articles](#))

Date de publication :

Dernière mise à jour :

Introduit avec la version 2 du Framework, les méthodes anonymes permettent de faire "pointer" un délégué sur un bout de code. Au travers de cet article, vous apprendrez comment utiliser les méthodes anonymes mais nous irons plus loin en comprenant leur mécanisme interne afin d'élucider certains comportements étranges...

- I - Pré-requis
- II - Introduction
- III - Des délégués aux méthodes anonymes
 - III.A - Cas de présentation
 - III-B - Méthode anonyme avec paramètre
 - III-C - Méthode anonyme avec paramètre par référence
 - III-D - Méthode anonyme et la généricité
- IV - Les méthodes anonymes et les évènements
- V - Les mystères des méthodes anonymes révélés
 - V-A - Au coeur des méthodes anonymes
 - V-B - Méthode anonyme qui accède à une variable locale à la méthode qui l'encapsule
 - V-C - Capture d'une variable locale par une méthode anonyme
- VI - Tout n'est pas possible avec les méthodes anonymes
- VI - Conclusion
- VIII - Remerciements
- IX - Liens

I - Pré-requis

Afin de suivre au mieux l'article, je vous conseille de vous munir des applications suivantes :

- Visual Studio 2005 avec C# (la version express gratuite est téléchargeable [ici](#)) afin de tester par vous même les exemples présentés
- L'outil **Reflector for .NET** permettant de voir le code pré-compilé contenu dans un assembly

II - Introduction

En Dotnet version 1, vous pouviez utiliser les délégués pour appeler une méthode, voire plusieurs méthodes les unes à la suite des autres (on comprend tout de suite le rapport qu'il y a avec les [FAQ pointeurs de fonctions du C/C++](#)). Avec Dotnet version 2, le concept des méthodes anonymes a été introduit afin de faire pointer des délégués non pas sur des méthodes mais directement sur du code "inline". Cela signifie qu'on va dire à un délégué qu'il doit pointer sur tel partie de code sans se soucier des signatures de méthodes.

Les évènements du Framework étant gérés par les délégués, vous verrez aussi comment faire la même chose mais avec des méthodes anonymes dans le but de simplifier l'écriture.

Enfin, nous irons dans le coeur des méthodes anonymes afin de mieux comprendre leur fonctionnement interne. Ne soyez pas pressés, vous en saurez plus bientôt...

III - Des délégués aux méthodes anonymes

III.A - Cas de présentation

Prenons un exemple tout simple : affichage de "Hello World" dans un MessageBox lors du clique sur un bouton.

Pour réaliser cette application avec un simple délégué, il nous faudra 2 choses:

- 1 Une méthode **AfficheHelloWorld** sans paramètre qui affiche "Hello World" dans un MessageBox
- 2 Un délégué **AfficheHelloWorld_Delegate**, sans paramètre aussi, qui "pointera" sur la méthode précédemment citée

Ce qui nous donne :

```
public partial class Form1 : Form
{
    //Déclaration du type délégué AfficheHelloWorld_Delegate
    delegate void AfficheHelloWorld_Delegate();

    AfficheHelloWorld_Delegate del_Hw = null;

    public Form1()
    {
        InitializeComponent();

        // Ajoute la méthode AfficheHelloWorld à l'instance du délégué AfficheHelloWorld_Delegate
        del_Hw = new AfficheHelloWorld_Delegate(AfficheHelloWorld);
    }

    // Méthode qui affiche Hello World dans un MessageBox
    void AfficheHelloWorld()
    {
        MessageBox.Show("Hello World !");
    }

    // Événement clique sur bouton
    // Affiche Hello World dans un MessageBox via un délégué
    private void bouton_TestDelegue_Click(object sender, EventArgs e)
    {
        // Exécute la méthode AfficheHelloWorld
        del_Hw();
    }
}
```

Pour un cas aussi simple, on aurait aimé ne pas passer son temps à créer une méthode juste pour exécuter un ligne...

Avec les méthodes anonymes, on va pouvoir contourner ce "problème" de la manière suivante :

```
public partial class Form1 : Form
{
    //Déclaration du type délégué AfficheHelloWorld_Delegate
    delegate void AfficheHelloWorld_Delegate();

    AfficheHelloWorld_Delegate del_Hw = null;
```

```
public Form1()
{
    InitializeComponent();

    // Ajoute directement au délégué le code à exécuter
    del_Hw = delegate { MessageBox.Show("Hello World !"); };
}

private void button_TestMethodAnonyme_Click(object sender, EventArgs e)
{
    // Exécute le code prédemment créé
    del_Hw();
}
}
```

Comme vous pouvez le constater, des accolades délimitent le code qui devra être exécuté par le délégué. Je me suis volontairement limité à une ligne de code mais rien ne nous empêche d'en écrire bien plus.

Maintenant, la question que vous devez vous poser est comment cela est-ce possible... Certains s'en doutent déjà mais passons en revue encore quelques exemples avant d'aborder ce sujet.

III-B - Méthode anonyme avec paramètre

Avec l'exemple suivant, vous pouvez vous apercevoir qu'une méthode anonyme peut accepter des paramètres de n'importe quel type tout comme elle peut renvoyer une valeur de n'importe quel type aussi :

```
class Program
{
    // Déclaration d'un type délégué avec paramètre et valeur de retour
    delegate string AffichePuissance2(int param_valeur);
    static void Main(string[] args)
    {
        AffichePuissance2 del_Puissance2 = delegate(int param_valeur)
        {
            int Resultat = param_valeur * param_valeur;
            return string.Format("{0} au carré donne {1}", param_valeur, Resultat);
        };

        Console.WriteLine(del_Puissance2(2));
        Console.WriteLine(del_Puissance2(4));
        Console.WriteLine(del_Puissance2(8));
    }
}
```

Avec comme résultat :

```
2 au carré donne 4
4 au carré donne 16
8 au carré donne 64
```

Grâce à l'Intelligence de Visual Studio, lorsqu'on veut exécuter le délégué, on est forcé d'entrer un paramètre de type **int** comme déclaré dans le type du délégué en question.

```
del_Puissance2 (  
string AffichePuissance2 (int param_valeur)
```

III-C - Méthode anonyme avec paramètre par référence

Un autre cas intéressant est le passage de paramètres par référence.


Modifions le code précédent afin d'obtenir :


```
class Program  
{  
    // Déclaration d'un type délégué avec paramètre et valeur de retour  
    delegate string AffichePuissance2ParReference(ref int param_valeur);  
    static void Main(string[] args)  
    {  
        AffichePuissance2ParReference del_Puissance2 = delegate(ref int param_valeur)  
        {  
            int copie_Valeur = param_valeur;  
            param_valeur = param_valeur * param_valeur;  
  
            return string.Format("{0} au carré donne {1}", copie_Valeur, param_valeur);  
        };  
  
        int valeur = 2;  
        Console.WriteLine(del_Puissance2(ref valeur));  
        Console.WriteLine(del_Puissance2(ref valeur));  
        Console.WriteLine(del_Puissance2(ref valeur));  
  
        Console.ReadLine();  
    }  
}
```

Avec comme résultat :

```
2 au carré donne 4  
4 au carré donne 16  
16 au carré donne 256
```

La valeur de la variable a donc bien été modifiée via la méthode anonyme après chaque appel.

 *A noter que les types des paramètres et de retour sont définis par le délégué auquel on assigne la méthode anonyme et non par la méthode anonyme elle même.*

 *L'utilisation du mot clé **params** est impossible avec les méthodes anonymes.*

III-D - Méthode anonyme et la généricité

Un autre cas avec les méthodes anonymes est l'utilisation des génériques. Les délégués supportant des arguments génériques, ces mêmes délégués pourront référencer une méthode anonyme ; la définition des types génériques devra se faire dans la définition de la méthode anonyme :

```
class Program
{
    delegate void DelegeGenerique<T>(T param_generique);

    static void Main(string[] args)
    {
        DelegeGenerique<int> del_Generique_Int = delegate(int param_generique)
        {
            Console.WriteLine("Le type est {0} avec comme valeur '{1}'",
param_generique.GetType().ToString(), param_generique.ToString());
        };

        DelegeGenerique<String> del_Generique_String = delegate(String param_generique)
        {
            Console.WriteLine("Le type est {0} avec comme valeur '{1}'",
param_generique.GetType().ToString(), param_generique.ToString());
        };

        del_Generique_Int(5);
        del_Generique_String("Hello");

        Console.ReadLine();
    }
}
```

Avec comme résultat :

```
Le type est System.Int32 avec comme valeur '5'
Le type est System.String avec comme valeur 'Hello'
```

IV - Les méthodes anonymes et les évènements

Tous les évènements du Framework passant par l'utilisation de délégués... Pourquoi ne pas utiliser les méthodes anonymes ?

Avec un simple délégué, on s'abonne comme ceci à l'évènement Click du composant Button (qui a dit double cliquer sur le bouton en mode design ?!) :

```
public Form1()
{
    InitializeComponent();

    bouton_TestClick.Click += new EventHandler(bouton_TestClick_Click);
}

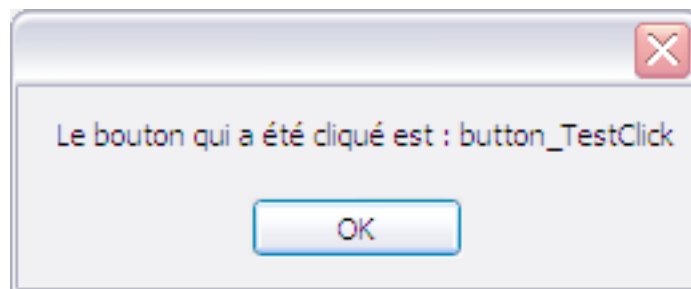
void bouton_TestClick_Click(object sender, EventArgs e)
{
    MessageBox.Show(string.Format("Le bouton qui a été cliqué est : {0}", ((Button)sender).Name));
}
```

Etant donné que nous avons vu comment passer des paramètres à une méthode anonyme, essayons la même chose dans ce cas ci :

```
public Form1()
{
    InitializeComponent();



    bouton_TestClick.Click += delegate(object sender, EventArgs e)
    {
        MessageBox.Show(string.Format("Le bouton qui a été cliqué est : {0}",
            ((Button)sender).Name));
    };
}
```

Avec comme résultat :



Ce qui est logique car quand on y réfléchit, tout évènement du Framework est géré par délégué.

V - Les mystères des méthodes anonymes révélés

 *Durant ce chapitre, je m'appuierai régulièrement sur l'outil  **Reflector** (de **Lutz Roeder**) permettant de parcourir du code compilé dans le but de décrire le comportement des méthodes anonymes dans des cas assez particuliers.*

V-A - Au coeur des méthodes anonymes

Commençons par un exemple tout simple déjà utilisé précédemment :

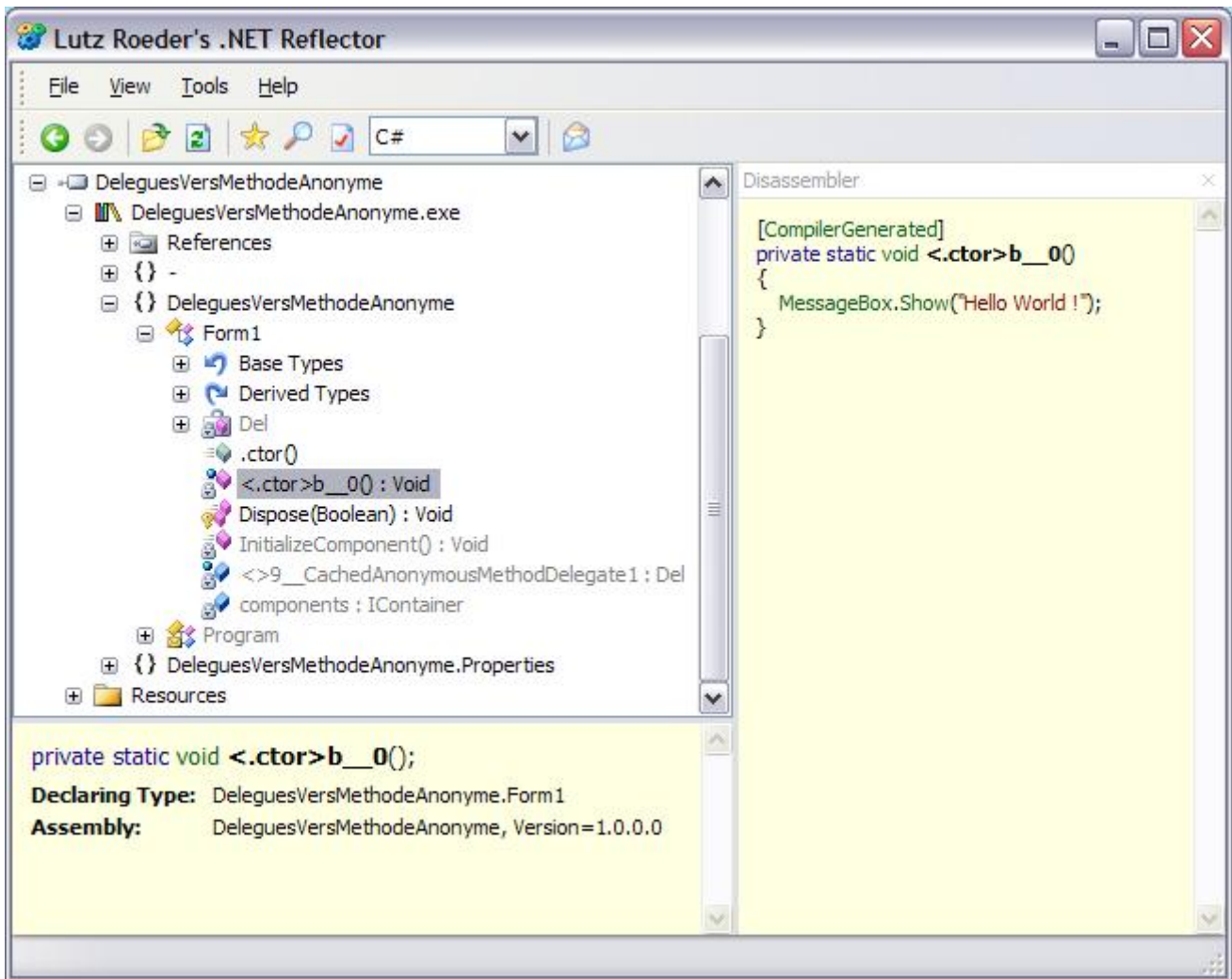
```
public partial class Form1 : Form
{
    //Déclaration du type délégué AfficheHelloWorld_Delegate
    delegate void Del();

    public Form1()
    {
        InitializeComponent();

        // Ajoute directement au délégué le code à exécuter
        Del del_Hello = delegate { MessageBox.Show("Hello World !"); };


        // Exécute le code prédemment créé
        del_Hello();
    }
}
```

Compilons le code et analysons l'exécutable à l'aide de **Reflector** :



Comme vous pouvez le constater, une méthode assez étrange a été créée (<.ctor>b__0). En analysant le code qui la compose, on s'aperçoit tout de suite qu'il s'agit de notre méthode anonyme.

Si donc pour nous développeur, une méthode anonyme ne représente qu'un bout de code, pour le CLR (Common Language Runtime) il s'agit d'une méthode nommée qui a été créée lors de de la compilation.

 *Pour les plus intrigués d'entre vous qui se demandent ce que vient faire <> dans un nom de méthode, cela permet tout simplement de dire que la méthode en question ne peut être directement appelée via le code.*

V-B - Méthode anonyme qui accède à une variable locale à la méthode qui l'encapsule

Dans l'exemple suivant, vous allez voir comment une variable locale va pouvoir être utilisée dans des méthodes anonymes :

```
class Program
{
    delegate void AfficheVariableLocale();
```

```
static AfficheVariableLocale monDelegue1;
static AfficheVariableLocale monDelegue2;

static void Main(string[] args)
{
    // Variable locale
    int maVariable = 5;

    monDelegue1 = delegate
    {
        maVariable = maVariable + 1;
        Console.WriteLine("Méthode anonyme1 : {0}", maVariable);
    };
    monDelegue2 = delegate
    {
        maVariable = maVariable + 1;
        Console.WriteLine("Méthode anonyme2 : {0}", maVariable);
    };

    monDelegue1();
    monDelegue2();
    Console.WriteLine("Méthode locale : {0}", maVariable);
    monDelegue1();
    monDelegue2();
    Console.WriteLine("Méthode locale : {0}", maVariable);
    monDelegue1();
    monDelegue2();
    Console.WriteLine("Méthode locale : {0}", maVariable);

    Console.ReadLine();
}
}
```

Avec comme résultat :

```
Méthode anonyme1 : 6
Méthode anonyme2 : 7
Méthode locale : 7
Méthode anonyme1 : 8
Méthode anonyme2 : 9
Méthode locale : 9
Méthode anonyme1 : 10
Méthode anonyme2 : 11
Méthode locale : 11
```

On s'aperçoit tout de suite que la variable a été partagée entre :

- La méthode locale
- La méthode anonyme **monDelegue1**
- La méthode anonyme **monDelegue2**

Pour comprendre ce phénomène étrange, jettons un oeil sur le code compilé avec Reflector :

Visual Studio .NET Reflector

Tools Help

C#

Disassembler

```
[CompilerGenerated]
private sealed class <>c__DisplayClass2
{
    // Fields
    public int maVariable;

    // Methods
    public void <Main>b__0()
    {
        this.maVariable++;
        Console.WriteLine("Méthode anonyme 1 : {0}", this.maVariable);
    }

    public void <Main>b__1()
    {
        this.maVariable++;
        Console.WriteLine("Méthode anonyme 2 : {0}", this.maVariable);
    }
}

Collapse Methods

private static void Main(string[] args)
{
    <>c__DisplayClass2 <>8__locals3 = new <>c__DisplayClass2();
    <>8__locals3.maVariable = 5;
    monDelegate1 = new AfficheVariableLocale(<>8__locals3.<Main>b__0);
    monDelegate2 = new AfficheVariableLocale(<>8__locals3.<Main>b__1);
    monDelegate1();
    monDelegate2();
    Console.WriteLine("Méthode locale : {0}", <>8__locals3.maVariable);
    monDelegate1();
    monDelegate2();
    Console.WriteLine("Méthode locale : {0}", <>8__locals3.maVariable);
    monDelegate1();
    monDelegate2();
    Console.WriteLine("Méthode locale : {0}", <>8__locals3.maVariable);
    Console.ReadLine();
}
```

class <>c__DisplayClass2

ConsoleApplication1.Program +<>c__DisplayClass2

ConsoleApplication1, Version=1.0.0.0

On peut constater plusieurs choses dans le code compilé :

- Nos deux méthodes anonymes se retrouvent encapsulées dans une classe appelée `<>c__DisplayClass2`
- La classe `<>c__DisplayClass2` a comme champs notre variable locale `maVariable`
- Le code de notre classe `Main` a été modifiée afin d'instancier un objet de la classe `<>c__DisplayClass2` à chaque appel
- La valeur de notre variable locale a directement été copiée dans le champs `maVariable` de la classe `<>c__DisplayClass2`

Au final, `maVariable` a été remplacé par :

- `this.maVariable` dans la classe `<>c__DisplayClass2`
- `<>8__locals3.maVariable` lorsque c'est la méthode `Main` elle même qui fait appel à sa variable locale

Ce qui explique ce partage de variable locale.

V-C - Capture d'une variable locale par une méthode anonyme

Cette fois-ci, nous allons créer une méthode qui renverra une instance d'une méthode anonyme utilisant une variable locale afin d'en observer le comportement :

```
class Program
{
    delegate void AfficheVariableLocale();

    static AfficheVariableLocale monDelegue1;
    static AfficheVariableLocale monDelegue2;

    static AfficheVariableLocale CreationDelegue()
    {
        // Variable locale
        int maVariable = 5;

        return delegate
        {
            maVariable = maVariable + 1;
            Console.WriteLine("Méthode anonyme : {0}", maVariable);
        };
    }

    static void Main(string[] args)
    {
        monDelegue1 = CreationDelegue();
        monDelegue2 = CreationDelegue();

        monDelegue1();
        monDelegue1();
        monDelegue1();
        monDelegue2();
        monDelegue2();
        monDelegue2();

        Console.ReadLine();
    }
}
```

Avec comme résultat :

```
Méthode anonyme : 6  
Méthode anonyme : 7  
Méthode anonyme : 8  
Méthode anonyme : 6  
Méthode anonyme : 7  
Méthode anonyme : 8
```

Il semblerait que chaque méthode anonyme a sa propre instance de la variable locale ; on dit que la variable a été capturée par la méthode anonyme.

Vérifions le code compilé avec une fois de plus **Reflector** :


Visual Studio .NET Reflector interface showing the disassembled code for an anonymous method. The left pane shows the class hierarchy with '<>c__DisplayClass1' selected. The right pane shows the disassembled code for the anonymous method, which is encapsulated in a private sealed class '<>c__DisplayClass1'. The code includes a field 'maVariable' and a method '<CreationDelegate>b__00' that increments 'maVariable' and writes it to the console. A static method 'CreationDelegate()' is also shown, which creates an instance of '<>c__DisplayClass1' and returns it.

On peut constater plusieurs choses dans le code compilé :

- Notre méthode anonyme se retrouve encapsulée dans une classe appelée **<>c__DisplayClass1**
- La classe **<>c__DisplayClass1** a comme champs notre variable locale **maVariable**

- Le code de notre classe **AfficheVariableLocale** a été modifiée afin d'instancier un objet de la classe **<>c__DisplayClass1** à chaque appel

Chaque méthode ayant sa propre instance de **<>c__DisplayClass1**, on comprend tout de suite que la variable locale n'est plus partagée.

 *De la même façon qu'il a été possible d'accéder à une variable locale à la méthode qui encapsule la méthode anonyme, il est possible aussi d'accéder :*

- *à un paramètre de la méthode qui encapsule la méthode anonyme*
- *ou encore à un champs de la classe qui définit la méthode qui encapsule la méthode anonyme*

VI - Tout n'est pas possible avec les méthodes anonymes

En effet, il y a bien quelque chose qui est impossible de faire avec les méthodes anonymes : ne plus référencer un bout de code.

Autant avec les méthodes nommées, on peut utiliser les opérateurs += et -= pour, respectivement, pointer et ne plus pointer sur une méthode donnée, autant avec les méthodes anonymes, c'est impossible à faire. Et la réponse est dans le titre, quand vous donnez la référence d'un bout de code à un délégué, c'est de manière anonyme avec donc aucune possibilité d'identifier ce bout de code par après.

```
delegate void Del();

static void f()
{
    Console.WriteLine("Code d'une fonction");
}
static void Main(string[] args)
{
    // Ajoute la référence de f
    Del monDelegate = new Del(f);
    // Ajoute la référence d'une méthode anonyme
    monDelegate += delegate { Console.WriteLine("Code d'une méthode anonyme"); };

    // Exécution de f et de la méthode anonyme
    monDelegate();

    // Retire la référence f
    monDelegate -= f;

    // Exécution de la méthode anonyme
    monDelegate();

    Console.ReadLine();
}
```

La seule alternative est de retirer toutes les références que le délégué contient en le mettant tout simplement à **null** :

```
monDelegate = null;
```

VI - Conclusion



Au travers de cet article, nous avons vu ce qu'était une méthode anonyme. Après quelques explications, vous avez appris à appliquer leur utilisation en reproduisant ce qui est déjà possible avec les méthodes nommées ; simples appels de méthodes, utilisation de paramètres et valeur de retour, les évènements, ...

Après cette partie pratique, la technique a fait place afin de comprendre le mécanisme des méthodes anonymes au travers d'exemples autrefois troublants.

VIII - Remerciements

Un grand merci à l'**équipe Dotnet** de Developpez.com et plus particulièrement à **fearyourself** pour les corrections.

IX - Liens

-  **Les méthodes anonymes** par Msdn
-  **Reflector** (de Lutz Roeder) permettant de parcourir du code compilé

