Introduction aux délégués en C#

par François Guillot (fguillot.developpez.com)

Date de publication : 2010-11-17

Dernière mise à jour : 2010-11-26

On constate ces dernières années l'éclosion d'une nouvelle génération d'informaticiens qui a appris à développer directement en langage managé. Bien que bénéficiant des nombreuses facilités qu'apporte un langage moderne comme le C#, celle-ci éprouve souvent quelques difficultés à appréhender une notion pourtant centrale de la programmation objet : les délégués. A contrario, les développeurs C et C++, déjà familiers de la notion de pointeur de méthode, sont naturellement plus enclins à cerner cette notion.

Cet article se propose donc d'expliquer aux débutants comment fonctionnent les délégués en C#, quelle est leur utilité, et quelle a été leur évolution avec les différentes versions du Framework .NET. Il peut également permettre aux développeurs plus expérimentés de réviser leurs connaissances sur ce vaste sujet.

I - .NET 1.0 et l'apparition des délégués....3
I-A - Démonstration par l'exemple....3
I-B - Exemple d'utilisation d'un délégué....4
I-C - L'inférence de type....7
II - .NET 2.0 et les méthodes anonymes....8
II-A - Le piège des fermetures....8
III - .NET 3.5 et les expressions lambda....11
IV - Les événements....13
IV-1 - Conserver la responsabilité de la levée de l'événement....14
IV-2 - Les accesseurs add et remove....14
IV-3 - Les événements dans les interfaces....16
V - Les délégués « clef en main »....18
V-A - Func....18
V-B - Action....18
V-C - Predicate....19
V-D - Converter....19
V-E - Comparison....19
V-F - EventHandler....20
V-G - Conclusion....21
VI - .NET 4.0, la covariance et la contravariance....22
VI-A - Le polymorphisme....22
VI-B - La covariance....22
VI-C - La contravariance....23
VI-D - Utilisation simultanée de la covariance et de la contravariance....23
VII - Conclusion....24
VII-A - Pour aller plus loin....24
VII-B - Remerciements....25

I - .NET 1.0 et l'apparition des délégués

Les délégués (en anglais « delegate ») étaient présents dès la première livraison du Framework .NET. Nous allons voir que leur principe de fonctionnement est resté globalement identique au fil des versions, tout en subissant quelques simplifications bienvenues.

I-A - Démonstration par l'exemple

Pour comprendre ce qu'apportent les délégués à la programmation avec le langage C#, nous allons étudier leur mécanisme avec un exemple concret nous permettant de bien comprendre leur utilité.

Imaginons que vous héritez d'une application dont l'une des tâches consiste à manipuler des tableaux de chaînes de caractères. Pour cela, elle dispose déjà d'une série de trois méthodes statistiques permettant de réordonner ces tableaux selon trois algorithmes de tri différents :

```
public static class SortingAlgorithms
{
    public static string[] BubbleSort(string[] data)
    {
        // Tri à bulle
        return data;
    }

    public static string[] QuickSort(string[] data)
    {
        // Tri rapide
        return data;
    }

    public static string[] InsertionSort(string[] data)
    {
        // Tri par insertion
        return data;
    }
}
```

En fonction du type de données à trier, et notamment de la taille du tableau, chacun de ces algorithmes est susceptible d'afficher de meilleures performances que les deux autres. Le développeur est donc laissé libre de choisir lequel conviendra le mieux en fonction des circonstances.

Par ailleurs, la méthode centrale de notre application est la suivante :

```
private static string[] ComputeArray(string[] data, SortingTypes sortingType)
{
    // Opérations sur le tableau

    // Tri du tableau
    switch (sortingType)
    {
        case SortingTypes.BubbleSort:
            data = SortingAlgorithms.BubbleSort(data);
            break;
        case SortingTypes.QuickSort:
            data = SortingAlgorithms.QuickSort(data);
            break;
        case SortingTypes.InsertionSort:
            data = SortingAlgorithms.InsertionSort(data);
            break;
    }

    // D'autres opérations sur le tableau

    return data;
}
```

Cette méthode prend en paramètre un tableau de chaînes de caractères, ainsi que la valeur d'une énumération lui permettant de savoir quel algorithme de tri lui appliquer le moment venu. Au cours du programme, d'autres opérations sont susceptibles d'être appliquées sur le tableau avant ou après le tri.

L'énumération définissant les différentes méthodes de tri du tableau est la suivante :

```
public enum SortingTypes
{
    BubbleSort,
    QuickSort,
    InsertionSort
}
```

On voit qu'à chaque algorithme de tri correspond une valeur de l'énumération.

Enfin voici la méthode d'entrée de notre programme de démonstration :

```
class Program
{
    static void Main(string[] args)
    {
        string[] data = new string[] { "toto", "titi", "tutu" };
        data = ComputeArray(data, SortingTypes.BubbleSort);
    }
}
```

Tout cela fonctionne correctement, et l'application fait son office. Mais voilà que votre chef de projet vient vous voir, et vous informe que des évolutions sont nécessaires. Les trois algorithmes de tri ne sont plus suffisants, et il vous faut maintenant en implémenter... trente-deux.

Quelles conséquences cela a-t-il sur notre application ? Tout d'abord, la classe statique SortingAlgorithms doit maintenant contenir trente-deux méthodes pour autant d'algorithmes différents. Cela n'a rien d'anormal car il faut bien que ces algorithmes soient implémentés quelque part. Par contre, l'énumération SortingTypes devra présenter trente-deux valeurs différentes, et surtout, notre switch/case permettant de sélectionner la bonne méthode de tri va grossir dans des proportions assez monstrueuses, altérant fortement sa lisibilité.

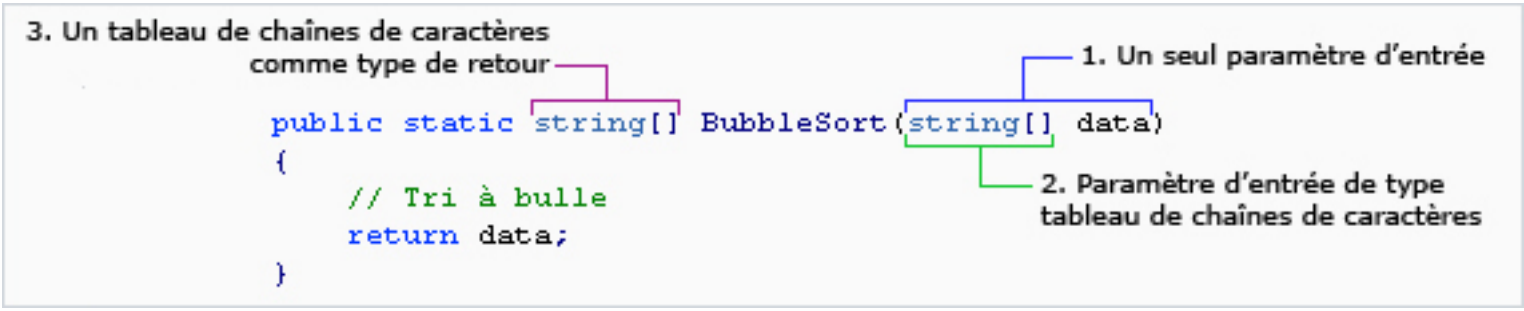
Mais après tout, faute de mieux, pourquoi pas ? Évidemment à ce stade vous avez déjà deviné que les délégués vont nous aider à modifier l'architecture de notre code pour nous sortir de ce mauvais pas.

I-B - Exemple d'utilisation d'un délégué

Mais d'abord, un constat s'impose. Quel est le point commun de toutes les méthodes de la classe statique SortingAlgorithms : BubbleSort, QuickSort et InsertionSort ?

Ces trois méthodes - et potentiellement les vingt-neuf autres que nous aurions à implémenter - ont toutes la même signature (on parle aussi de « prototype »). Leurs noms sont bien entendu différents, mais elles ont toutes :

1. le même nombre de paramètres d'entrée (ici un seul) ;
2. des paramètres d'entrée du même type (un tableau de chaînes de caractères) ;
3. le même type de retour (également un tableau de chaînes de caractères).



Cette ressemblance n'est pas un hasard, puisque ces trois méthodes sont conçues pour effectuer la même tâche : trier le tableau qui leur est passé en paramètre, bien que chacune le fasse d'une façon différente.

C'est ce point commun que l'utilisation d'un délégué va précisément nous permettre d'abstraire.

Tout d'abord observons la déclaration d'un délégué :

```
public delegate string[] SortingMethod(string[] data);
```

Les délégués, au même titre que les classes, les énumérations, ou les structures sont ce que l'on appelle communément des « first-class citizens », soit des membres de plus haut niveau qui, lorsqu'on les déclare, créent un nouveau type à part entière.

La déclaration d'un délégué, grâce au mot-clef « delegate », peut se faire soit directement dans un espace de nom (mot-clef « namespace »), soit à l'intérieur d'une classe. En fonction de l'un ou l'autre, les modificateurs d'accès que l'on pourra leur appliquer varieront, mais seront les mêmes que pour une classe. Admettons ici que nous avons déclaré ce délégué au niveau de l'espace de nom du programme.

La déclaration d'un délégué ne correspond ni plus ni moins qu'à une signature de méthode. En effet, on retrouve ici le même nombre de paramètres, du même type, et le même type de retour que nos trois méthodes de tri.

Notez au passage que la notion de surcharge de méthode ne concerne pas les délégués. Contrairement aux méthodes, on ne peut avoir deux délégués portant le même nom et déclarant des signatures différentes. Ce concept leur est totalement étranger.

Dans ce délégué SortingMethod, nous allons pouvoir « enregistrer » des méthodes respectant sa signature. On dit plus communément que le délégué va encapsuler une ou plusieurs de ces méthodes.

Notre programme devient alors :

```
namespace DelegateProgram
{
    public delegate string[] SortingMethod(string[] data);

    class Program
    {
        static void Main(string[] args)
        {
            string[] data = new string[] { "toto", "titi", "tutu" };
            SortingMethod sortingMethod = new SortingMethod(SortingAlgorithms.BubbleSort);
            ComputeArray(data, sortingMethod);
        }

        private static string[] ComputeArray(string[] data, SortingMethod sortingMethod)
        {
            // Opérations sur le tableau

            // Tri du tableau
            data = sortingMethod(data);
```

```
            // D'autres opérations sur le tableau

            return data;
        }
    }
}
```

On constate tout de suite que plusieurs changements ont été opérés.

Tout d'abord, notre énumération SortingTypes, désormais inutile, a totalement disparu. À sa place, notre méthode de traitement principale ComputeArray attend maintenant un paramètre du type de notre délégué SortingMethod. Pour appeler cette méthode, nous instancions donc un objet de type SortingMethod, au constructeur duquel nous passons une référence vers une des méthodes de tri - ici BubbleSort.

```
SortingMethod sortingMethod = new SortingMethod(SortingAlgorithms.BubbleSort);
ComputeArray(data, sortingMethod);
```

L'expression « passer une référence » est importante. Remarquez que lorsque nous déclarons la méthode BubbleSort dans le constructeur du délégué SortingMethod, son nom n'est pas suivi de parenthèses « () ». En effet, à ce stade nous n'exécutons pas encore la méthode BubbleSort ; nous nous contentons de la pointer afin que le délégué en garde une référence. Cette notation sans parenthèses est ce que l'on appelle un « groupe de méthodes » (en anglais « method group »), avec « méthodes » au pluriel car BubbleSort pourrait tout à fait avoir plusieurs surcharges, lesquelles seraient toutes automatiquement incluses.

En l'occurrence, le délégué SortingMethod n'encapsulera que les surchages de méthode respectant scrupuleusement la signature qu'il définit. Toute tentative de lui faire encapsuler une méthode n'ayant aucune surcharge dont la signature est conforme se soldera par l'impossibilité de compiler le programme. Ici tout va bien puisque la seule surcharge de la méthode statique BubbleSort respecte bien la signature définie par le délégué SortingMethod.

Autre changement notable dans notre programme : auparavant notre méthode principale ComputeArray recevait en paramètre une valeur de l'énumération qui lui indiquait quelle méthode de tri elle devait appliquer sur le tableau. Grâce au délégué SortingMethod, nous lui passons maintenant directement une référence vers la méthode elle-même. Ne reste alors plus qu'à exécuter le délégué, et se faisant, la méthode qu'il encapsule.

```
// Tri du tableau
data = sortingMethod(data);
```

C'est à ce moment-là, et à ce moment-là seulement, que nous exécutons réellement l'algorithme de tri BubbleSort. Grâce au délégué nous avons donc « enregistré » cette méthode afin de retarder son exécution.

Notez que la méthode ComputeArray exécute le délégué qui lui est passé en paramètre en ignorant totalement quelle méthode de tri celui-ci encapsule. Mais nous qui avons conçu le programme savons bien que c'est BubbleSort qui est exécuté.

Il est aussi possible grâce à l'opérateur « += » de faire référencer au délégué plusieurs méthodes. On parle alors de délégué « multicast ».

```
SortingMethod sortingMethod = new SortingMethod(SortingAlgorithms.BubbleSort);
sortingMethod += SortingAlgorithms.QuickSort;
sortingMethod += SortingAlgorithms.InsertionSort;
```

Lors de l'exécution du délégué, les trois méthodes de tri seront exécutées tour à tour, dans l'ordre où elles ont été référencées. À la sortie nous récupérerons systématiquement ce que renvoie la dernière méthode exécutée, ici en l'occurrence InsertionSort.

Il est de même possible grâce à l'opérateur « -= » de déréférencer une méthode :

```
SortingMethod sortingMethod = new SortingMethod(SortingAlgorithms.BubbleSort);
sortingMethod += SortingAlgorithms.QuickSort;
sortingMethod -= SortingAlgorithms.BubbleSort;
```

Ici sortingMethod ne référence que la méthode QuickSort, puisque BubbleSort a été référencée puis déréférencée.

Finalement, nous avons pu nous débarrasser à la fois de notre énumération, mais aussi et surtout de notre switch/case qui promettait de devenir rapidement encombrant.

I-C - L'inférence de type

Il est possible de simplifier encore davantage la syntaxe de notre programme. En effet, le compilateur C# est capable « d'inférer », c'est-à-dire de déterminer tout seul le type d'un délégué, grâce au type de la variable qui va l'accueillir. Nous pouvons dès lors initialiser la variable sortingMethod de la façon suivante :

```
SortingMethod sortingMethod = SortingAlgorithms.BubbleSort;
```

Nous pouvons même nous passer totalement de cette variable :

```
ComputeArray(data, SortingAlgorithms.BubbleSort);
```

Encore une fois, cette opération n'est possible que parce que la signature d'au moins une des surcharges de la méthode BubbleSort - qui ici n'en a qu'une - respecte celle définie par le délégué SortingMethod. Le compilateur C# fait la conversion entre les deux automatiquement.

Dès lors, appeler notre méthode principale ComputeArray n'a jamais été aussi simple.

II - .NET 2.0 et les méthodes anonymes

Avec le Framework .NET 2.0 est arrivée une nouvelle forme de déclaration de délégués : les méthodes anonymes. En fait il ne s'agit ni plus ni moins que de déclarer directement le code que devra référencer un délégué, sans avoir à créer une méthode pour le contenir.

Toujours dans l'esprit de notre exemple du tri d'un tableau de chaînes de caractères, imaginons que nous souhaitions trier celui-ci selon un algorithme à usage unique, c'est-à-dire que l'on utilisera à un seul endroit de l'application, et qui n'a donc pas besoin d'être isolé dans sa propre méthode. La déclaration de notre instance de délégué pourrait alors ressembler à ceci :

```
SortingMethod sortingMethod = new SortingMethod(delegate(string[] data)
                                    {
                                        // Tri arbitraire
                                        return data;
                                    });
```

Nous créons « à la volée » (on dit aussi « inline ») une méthode que va référencer le délégué. Cette méthode ne porte pas de nom, d'où le terme de « méthode anonyme ». Pourtant ce code sera exécuté comme n'importe quelle autre méthode lors de l'appel du délégué.

Remarquez que nous respectons bien la signature du délégué SortingMethod. En effet, notre méthode anonyme attend un seul paramètre d'entrée de type tableau de chaînes de caractères qui fait aussi office de type de retour puisque celle-ci le renvoie (théoriquement après l'avoir trié).

Nous avons ici donné arbitrairement le nom de « data » au paramètre d'entrée, ce qui nous permet ensuite d'y accéder dans le corps de la méthode anonyme, symbolisé par la partie entre accolades « {} ».

En outre, grâce aux capacités d'inférence de type dont nous avons parlé plus tôt, nous pouvons aussi déclarer notre méthode anonyme de la façon suivante :

```
SortingMethod sortingMethod = delegate(string[] data)
                              {
                                  // Tri arbitraire
                                  return data;
                              };
```

Ou encore la passer directement à la méthode ComputeArray :

```
ComputeArray(data, delegate(string[] data)
                   {
                       // Tri arbitraire
                       return data;
                   });
```

II-A - Le piège des fermetures

Les délégués anonymes peuvent sembler à la fois simples et pratiques, pourtant ils introduisent un piège mortel dans lequel de nombreux débutants sont tombés : les fermetures (ou clôtures ; en anglais, « closure »).

La notion de fermeture indique que la portée d'une variable change lorsqu'on la passe à une méthode anonyme. Ce n'est pas clair ? Pour mieux comprendre, étudions un nouvel exemple :

```
public delegate bool ClosureDelegate();

class Program
{
    static void Main(string[] args)
```

```
    {
        bool test = false;

        ClosureDelegate closure = delegate() { return test; };
        test = true;

        Console.WriteLine(closure());
    }
}
```

Nous déclarons un délégué ClosureDelegate qui définit une signature de méthode ne prenant pas de paramètre d'entrée et retournant un booléen. Puis dans notre programme nous instancions ce délégué, et le faisons encapsuler une méthode anonyme qui retourne la valeur d'une variable. Notez que cette variable nommée test a été déclarée et initialisée en dehors de la méthode anonyme.

Or justement, entre le moment où nous instancions le délégué et celui où nous l'exécutons, la valeur de notre variable test a changé. Du coup, que va bien pouvoir retourner l'exécution de notre délégué ? La valeur de la variable telle qu'elle était au moment de la création du délégué, ou celle au moment de son exécution ?

La question est d'autant plus ambigüe qu'on peut légitimement se demander quel comportement l'auteur de ces quelques lignes de code pouvait s'attendre à observer.

Voici ce qui s'affiche dans la console si l'on exécute notre programme :

C'est donc la valeur de la variable au moment de l'exécution du délégué qui est retournée.

Pourquoi ce résultat ? Parce que la variable test est déclarée en dehors de la portée (on dit aussi « scope ») de la méthode anonyme. C'est cette portée que l'on nomme précisément la « fermeture », car celle-ci est partagée avec celle du code qui la déclare, là où une vraie fonction aurait sa propre portée.

C'est ce partage de la portée entre la méthode anonyme et le code qui la déclare qui fait qu'au lieu de conserver la valeur de la variable « test » au moment de sa création, le délégué a accès à sa dernière valeur qui est susceptible de varier comme dans notre exemple. Le problème est donc que le compilateur n'a aucun moyen de déterminer si vous souhaitez ou non qu'il suive les changements éventuels de valeur de la variable.

Dans le cas présent, si notre souhait est que le délégué retourne la valeur de la variable « test » au moment de son instanciation, il n'y a qu'une seule solution :

```
static void Main(string[] args)
{
    bool test = false;

    bool temp = test; // Ne pas modifier!
    ClosureDelegate closure = delegate() { return temp; };

    test = true;

    Console.WriteLine(closure());
}
```

Créer une nouvelle variable intermédiaire permet à la fois d'utiliser la valeur de « test » dans la méthode anonyme, mais aussi de modifier celle-ci par la suite sans risquer d'altérer le comportement du délégué à l'exécution. Évidemment la condition sine qua non est que la valeur de la nouvelle variable ne soit pas non plus modifiée entretemps.

Nous obtenons alors le résultat attendu :

La notion de fermeture est assez pointue et peut être difficile à appréhender. La leçon à retenir est qu'il faut se méfier des méthodes anonymes, et penser à vérifier la portée des variables que l'on utilise dans le corps de celles-ci.

III - .NET 3.5 et les expressions lambda

Le Framework .NET 3.5 a apporté de nombreuses nouveautés, notamment en terme de syntaxe. Les méthodes anonymes s'en sont vues largement simplifiées grâce aux expressions lambda qui permettent de les déclarer de manière encore plus compacte qu'auparavant.

Mais reprenons notre méthode anonyme de tout à l'heure :

```
SortingMethod sortingMethod = new SortingMethod(delegate(string[] data)
                                    {
                                        // Tri arbitraire
                                        return data;
                                    });
```

Écrit sous la forme d'une expression lambda, ce code devient :

```
SortingMethod sortingMethod = new SortingMethod((string[] data) =>
                                    {
                                        // Tri arbitraire
                                        return data;
                                    });
```

À priori peu de choses ont changé. La seule différence est que le mot-clef « delegate » a disparu, et qu'un nouvel opérateur, l'opérateur lambda « => » qui se lit « conduit à », est apparu. Mais ceci ne constitue qu'une première étape, car les expressions lambda vont nous permettre d'utiliser toute la puissance du compilateur C# pour simplifier notre déclaration.

Tout d'abord, puisque notre expression lambda représente une méthode anonyme, le compilateur C# est capable d'inférer automatiquement le type de ses paramètres d'entrée et son type de sortie à partir du type de délégué que nous indiquons vouloir instancier. Ce qui nous permet d'ores et déjà de simplifier notre code ainsi :

```
SortingMethod sortingMethod = new SortingMethod((data) =>
                                    {
                                        // Tri arbitraire
                                        return data;
                                    });
```

Ensuite, en fonction du nombre de paramètres d'entrée, différentes règles peuvent s'appliquer :

1. s'il n'y a aucun paramètre d'entrée, il faut l'indiquer obligatoirement par des parenthèses vides « () » ;
2. s'il n'y a qu'un seul et unique paramètre d'entrée, les parenthèses sont facultatives ;
3. s'il y a plusieurs paramètres d'entrée, ceux-ci doivent être encadrés par des parenthèses « () » et séparés par des virgules.

Voici plusieurs exemples appliquant ces règles :

```
// Expression lambda sans paramètre d'entrée
() =>  { return string.Empty; };

// Expression lambda avec un seul paramètre d'entrée
data => { return true; };

// Expression lambda avec plusieurs paramètres d'entrée
(data, data2) => { return 2 + 2; };
```

Par ailleurs, une autre règle peut s'appliquer à la déclaration du corps de la méthode anonyme. Si celle-ci ne contient qu'une seule et unique ligne, possibilité nous est offerte de supprimer les accolades « {} », le mot-clef « return » ainsi que le point virgule de fin de déclaration « ; ». Ainsi, si l'on retire dans notre exemple la ligne de commentaire - qui n'est pas à proprement parler une ligne de code - nous pouvons appliquer cette règle afin d'obtenir :

```
SortingMethod sortingMethod = new SortingMethod(data => data);
```

Bien entendu, cette expression lambda n'est là qu'à titre d'exemple et n'a pas grande utilité, puisqu'elle se contente de retourner inchangé le tableau qu'on lui aura passé en paramètre. En tout état de cause nous sommes encore parvenus à simplifier la syntaxe de notre méthode anonyme.

Enfin, comme nous l'avons déjà fait précédemment, nous allons pouvoir utiliser les capacités d'inférence du compilateur C# afin de nous affranchir de la nécessité d'instancier explicitement le délégué SortingMethod. D'ailleurs, en matière d'inférence de type le Framework .NET 3.5 a aussi introduit le mot-clef « var » qui permet de typer une variable implicitement si elle est déclarée en même temps qu'elle est instanciée.

Pour tester si vous avez bien compris ce principe d'inférence de type, pouvez-vous dire laquelle de ces trois lignes de code ne compilera pas ?

```
// 1.
SortingMethod sortingMethod = data => data;

// 2.
var sortingMethod = data => data;

// 3.
var sortingMethod = new SortingMethod(data => data);
```

Avez-vous trouvé ? C'est le proposition n°2 qui ne compilera pas. En effet, le fait d'utiliser le mot-clef « var » et dans le même temps de ne pas typer explicitement la déclaration de la méthode anonyme a l'effet suivant : le compilateur C# est incapable d'inférer automatiquement le type de la variable sortingMethod car il ne sait tout simplement pas quel est son vrai type. En revanche, les deux autres propositions fonctionneront parfaitement.

Pour en finir avec les expressions lambda, n'oubliez pas que celles-ci étant de simples méthodes anonymes le piège des fermetures y est plus que jamais d'actualité. Sachez aussi que l'utilisation des expressions lambda dans le cadre des délégués ne représente qu'une petite partie de leur utilité, celles-ci permettant aussi la création d'arbres d'expressions.

IV - Les événements

On ne peut pas parler des délégués dans le Framework .NET sans parler des événements, les deux étant intimement liés. Grâce aux événements (en anglais « event »), une classe a la possibilité d'effectuer du « push », c'est-à-dire d'informer le reste du code qu'il se passe quelque chose de particulier afin que celui-ci réagisse en conséquence.

Prenons un cas concret : imaginons que vous deviez concevoir une classe TrainStation représentant une gare ferroviaire. Cette classe va avoir la responsabilité d'informer de l'arrivée d'un train, grâce à un événement TrainArrival.

Voici à quoi elle ressemble :

```
public delegate void TrainArrivalEventHandler(object sender, EventArgs e);

public class TrainStation
{
    public event TrainArrivalEventHandler TrainArrival;
}
```

Remarquons tout d'abord que la création d'un événement est soumise préalablement à celle d'un délégué ayant une signature bien précise : pas de type de retour (« void »), et deux paramètres d'entrée, l'un de type object qui permettra d'indiquer quel objet a levé l'événement, et l'autre de type EventArgs - ou d'une classe en héritant - qui va encapsuler les arguments de l'événement.

Ceci est une recommandation des bonnes pratiques dans le Framework .NET
Vous pouvez tout à fait utiliser un délégué totalement différent pour créer votre événement ; le code compilera sans aucun problème. Mais vous vous exposerez à des ennuis à plus ou moins long terme, notamment si vous utilisez des librairies qui respectent ces bonnes pratiques. C'est pourquoi il est fortement recommandé de respecter cette consigne.

La déclaration de l'événement en lui-même se fait sous la forme d'un champ (en anglais « field »), avec le mot-clef « event », et en indiquant quel délégué sera chargé d'encapsuler les méthodes qui vont s'abonner à cet événement. Ce délégué servant de gestionnaire d'événement, on le nomme traditionnellement du même nom que l'événement lui-même en le suffixant avec « EventHandler », comme c'est le cas ici pour TrainArrivalEventHandler.

Notre programme va maintenant pouvoir instancier la classe TrainStation et s'abonner à son événement TrainArrival :

```
class Program
{
    static void Main(string[] args)
    {
        TrainStation trainStation = new TrainStation();
        trainStation.TrainArrival += OnTrainArrival;
    }

    private static void OnTrainArrival(object sender, EventArgs e)
    {
        Console.WriteLine("Un train est entré en gare.");
    }
}
```

On remarque tout de suite que s'abonner à un événement se fait exactement de la même façon que s'abonner à un délégué. D'ailleurs la méthode OnTrainArrival que nous utilisons respecte bien la signature du délégué TrainArrivalEventHandler. Nous pourrions dans le même esprit faire s'abonner plusieurs méthodes à l'événement, et même utiliser des méthodes anonymes. Le fait de préfixer le nom de cette méthode par « On » et de lui donner le même nom que l'événement est encore une convention de nommage chère au .NET.

Pour que la méthode OnTrainArrival soit exécutée, il faut que la classe TrainStation exécute l'événement. En fait on dit qu'elle va lever l'événement (en anglais « to raise ») ou l'invoquer (en anglais « to invoke »).

```
public class TrainStation
{
    public event TrainArrivalEventHandler TrainArrival;

    public void RaiseTheEvent()
    {
        if (TrainArrival != null)
            TrainArrival(this, EventArgs.Empty);
    }
}
```

Lever l'événement revient simplement à exécuter le délégué en lui passant l'objet à l'origine de cette levée (généralement la classe elle-même, mais cela peut-être un contrôle dans le cas d'une application graphique), ainsi que les arguments de l'événement (ici nous utilisons la propriété statique EventArgs.Empty pour indiquer qu'il n'y en a aucun).

Remarquez que si aucune méthode n'était abonnée à notre événement celui-ci serait null, et tenter de le lever provoquerait une exception, c'est pourquoi il faut systématiquement le tester préalablement.

Observons le résultat de la levée de l'événement directement par le programme lui-même, chose possible parce que la méthode RaiseTheEvent de la classe TrainStation est publique.

```
static void Main(string[] args)
{
    TrainStation trainStation = new TrainStation();
    trainStation.TrainArrival += OnTrainArrival;
    trainStation.RaiseTheEvent();
}
```

Finalement un événement se comporte exactement comme le délégué dont il dépend. Pourtant nous allons voir qu'il apporte quelques mécanismes supplémentaires.

IV-1 - Conserver la responsabilité de la levée de l'événement

Tout d'abord, si nous utilisions directement un délégué comme champ public de la classe, le programme aurait la possibilité de s'abonner au délégué, mais également de l'exécuter. Un événement permet à la classe de conserver la responsabilité de la levée de l'événement, et donc de l'exécution du délégué. Dans notre dernier exemple, si la méthode RaiseTheEvent était privée, seule la classe TrainStation serait en mesure de l'appeler, et donc de lever l'événement TrainArrival.

IV-2 - Les accesseurs add et remove

Autre avantage des événements, ils disposent de deux accesseurs « add » et « remove », de la même manière que les propriétés disposent de « get » et « set ». Grâce à ces accesseurs, il est possibilité d'effectuer des actions particulières lorsqu'une méthode s'abonne ou se désabonne d'un événement.

Mais l'utilisation de ces accesseurs va aussi nécessiter quelques aménagements dans notre classe TrainStation :

```
public class TrainStation
```

```
{
    private TrainArrivalEventHandler _trainArrival;

    public event TrainArrivalEventHandler TrainArrival
    {
        add
        {
            Console.WriteLine("Quelqu'un s'est abonné à l'événement.");
            _trainArrival += value;
        }

        remove
        {
            Console.WriteLine("Impossible de se désabonner de l'événement.");
        }
    }

    public void RaiseTheEvent()
    {
        if (_trainArrival != null)
            _trainArrival(this, EventArgs.Empty);
    }
}
```

Tout d'abord, notez que l'utilisation de l'un de ces accesseurs oblige à utiliser l'autre, quitte à ce que celui-ci soit vide, sinon le code ne compilera pas. Notez aussi qu'il vous revient d'encapsuler dans le délégué la méthode qui souhaite s'abonner à l'événement, en utilisant le mot-clef « value ». Ici en l'occurrence nous ne désencapsulons pas les méthodes qui pourraient en faire la demande. Ainsi le programme suivant :

```
static void Main(string[] args)
{
    TrainStation trainStation = new TrainStation();
    trainStation.TrainArrival += OnTrainArrival;
    trainStation.TrainArrival -= OnTrainArrival;
    trainStation.RaiseTheEvent();
}
```

Donne ce résultat :

Comme prévu, une fois abonnée à l'événement, la méthode OnTrainArrival n'a pas été en mesure de s'en désabonner.

Mais il y a autre chose qui a changé dans notre classe TrainStation. Nous y conservons maintenant une instance du délégué TrainArrivalEventHandler, dans le champ privé _trainArrival, et c'est maintenant cette instance que nous exécutons en lieu et place de l'événement lui-même, dans la méthode RaiseTheEvent.

En effet, l'utilisation des accesseurs add et remove oblige à passer par une véritable instance de délégué, chose qui était implicite jusqu'à maintenant. Cette obligation est due aux accesseurs grâce auxquels nous avons maintenant la possibilité d'encapsuler les méthodes souhaitant s'abonner à l'événement dans plusieurs délégués différents.

```
public class TrainStation
{
    private TrainArrivalEventHandler _trainArrivalDecember;
```

```
    private TrainArrivalEventHandler _trainArrivalOtherMonths;

    public event TrainArrivalEventHandler TrainArrival
    {
        add
        {
            if (DateTime.Today.Month == 12)
                _trainArrivalDecember += value;
            else
                _trainArrivalOtherMonths += value;
        }

        remove
        {
            if (DateTime.Today.Month == 12)
                _trainArrivalDecember -= value;
            else
                _trainArrivalOtherMonths -= value;
        }
    }

    public void RaiseTheEvent(bool raiseDecember)
    {
        if (_trainArrivalOtherMonths != null)
            _trainArrivalOtherMonths(this, EventArgs.Empty);

        if (raiseDecember && _trainArrivalDecember != null)
            _trainArrivalDecember(this, EventArgs.Empty);
    }
}
```

Ici par exemple nous décidons tout à fait arbitrairement d'encapsuler les méthodes dans deux délégués en fonction de la date courante. Si nous sommes au mois de décembre (DateTime.Today.Month == 12), les méthodes seront encapsulées dans le délégué _trainArrivalDecember ; sinon elles le seront dans le délégué _trainArrivalOtherMonths. Ce scénario serait envisageable dans une application conçue pour tourner sans interruption pendant plusieurs mois. Nous modifions également la méthode RaiseTheEvent, qui prend maintenant en paramètre un booléen lui indiquant si elle doit ou non exécuter le délégué correspondant au mois de décembre. Notez que la classe TrainStation conserve bien la responsabilité de la levée de l'événement.

IV-3 - Les événements dans les interfaces

Pour rappel, une interface permet d'abstraire les signatures de certains éléments d'une classe. En l'occurrence : depuis le Framework .NET 3.5, les propriétés et les indexeurs, et dès la version 1.0, les méthodes et les événements. C'est évidemment cette dernière possibilité qui nous intéresse.

En effet, une interface n'est pas en mesure d'abstraire un délégué, étant donné son statut de « first-class citizen ». Aussi sans l'existence des événements, nous ne serions pas en mesure d'abstraire notre classe TrainStation de la sorte :

```
public interface ITrainStation
{
    event TrainArrivalEventHandler TrainArrival;
}

public class TrainStation : ITrainStation
{
    public event TrainArrivalEventHandler TrainArrival;

    // Reste du code omis par volonté de clarté
}
```

Dorénavant, si nous sommes amenés à manipuler un objet de type ITrainStation, nous sommes assurés que celui-ci remplit le contrat établi par l'interface et implémente bien un événement TrainArrival. Ce mécanisme peut nous

permettre d'établir des scénarios où l'implémentation d'ITrainStation sera cachée à une partie du code, lors de l'utilisation de services ou du principe d'injection de dépendances par exemple.

V - Les délégués « clef en main »

Tout au long de l'évolution du Framework .NET ont été introduits des délégués que l'on peut qualifier de « clef en main », car ils permettent de se débarrasser du principe même de déclaration de délégués.

En effet, avec un peu d'habitude, le mécanisme offert par les délégués devient vite incontournable, et il devenait pénible de devoir déclarer pour chaque cas nécessitant leur utilisation un type de délégué approprié. Souvenez-vous de cette ligne vue plus haut :

```
public delegate string[] SortingMethod(string[] data);
```

Grâce aux délégués clef en main, nous allons pouvoir nous affranchir de cette déclaration.

V-A - Func

Un Func représente un délégué ayant un type de retour, et jusqu'à quatre paramètres d'entrée (seize dans le Framework .NET 4.0). On utilise la généricité pour indiquer ces différents types, de la façon suivante :

```
Func<string, int, byte[], short, bool> func = new Func<string, int, byte[], short, bool>((str, i, b,
 sh) => true);
```

Dans cet exemple, le type de retour est un booléen. En effet, retenez que le type de retour est toujours le dernier paramètre générique (d'ailleurs nous voyons que l'expression lambda renvoie « true »). La méthode anonyme prend donc comme paramètres, et dans l'ordre : un string, un nombre entier codé sur 32 bits (« int »), un tableau d'octets, et un nombre entier codé sur 16 bits (« short »). Nous avons nommé ceux-ci arbitrairement « str », « i », « b » et « sh », deux paramètres ne pouvant évidemment pas porter le même nom.

Comme déjà vu, il est aussi possible d'écrire :

```
Func<string, int, byte[], short, bool> func = (s, i, b, sh) => true;
```

Un Func déclare toujours un type de retour, mais il peut tout à fait n'accepter aucun paramètre :

```
// L'exécution de ce délégué renverra toujours 12
Func<int> func = () => 12;
```

Vous vous souvenez de la méthode ComputeArray de notre programme de tri de tableaux de chaînes de caractères ? Au lieu de la déclarer comme nous l'avions fait :

```
private static string[] ComputeArray(string[] data, SortingMethod sortingMethod)
```

Nous pourrions grâce au délégué Func la déclarer de la sorte, nous épargnant ainsi la nécessité de déclarer le délégué SortingMethod :

```
private static string[] ComputeArray(string[] data, Func<string[], string[]> sortingMethod)
```

V-B - Action

Un Action représente un délégué n'ayant pas de type de retour, et jusqu'à quatre paramètres d'entrée (seize dans le Framework .NET 4.0).

L'existence d'Action est nécessaire, étant donné que le mot-clef « void », qui permet d'indiquer dans une signature de méthode que celle-ci ne retourne rien, n'est bel et bien qu'un mot-clef et pas un type en soi. Il ne peut donc pas être utilisé comme paramètre générique.

```
Func<string, void> func; // Cette déclaration est impossible
```

Action est donc strictement équivalent à Func, à ceci près qu'il n'a donc jamais de type de retour. En voici un exemple :

```
Action<IList<string>> action = new Action<IList<string>>(l => l.Clear());
```

Exécuter cet Action sur une collection de chaînes de caractères aura pour effet de vider celle-ci de son contenu. On voit qu'en dehors d'effectuer une action, l'exécution du délégué ne renverra rien.

V-C - Predicate

En informatique, on appelle communément « prédicat » toute expression qui, lorsqu'on l'évalue, renvoie un booléen. Ainsi lorsque vous écrivez ceci :

```
if (variable == 2)
```

Le contenu du if est précisément ce que l'on nomme un prédicat.

Un Predicate est comme un Func qui renverrait systématiquement un booléen. Par ailleurs, celui-ci ne peut prendre qu'un seul paramètre d'entrée, dont le type vous échoit par la généricité. Par exemple :

```
Predicate<int> predicate = new Predicate<int>(i =>
                                    {
                                        int j = i + 12;
                                        return j == 37;
                                    });
```

À l'exécution, ce délégué additionnera douze au nombre entier qui lui est passé en paramètre, puis testera que le résultat est bien égal à trente-sept. Remarquez que pour l'occasion nous faisons appel à une expression lambda dont le corps, encadré par des accolades « {} », s'étend sur plusieurs lignes.

V-D - Converter

Un Converter permet de convertir un objet d'un type vers un autre. Il accepte deux paramètres génériques : le type source, qui fait office de paramètre d'entrée, et le type destination, qui fait office de type de retour. Par exemple :

```
Converter<int, long> converter = new Converter<int, long>(i => (long)i);
```

Ce Converter se contente de prendre un nombre entier codé sur 32 bits en paramètre d'entrée, et de le caster explicitement vers le type de retour, un nombre entier codé sur 64 bits (ce qui est sans douleur pour le compilateur).

Évidemment d'autres scénarios de conversion plus exotiques sont envisageables. Il vous incombe de définir à chaque fois comment la conversion se fera d'un type vers l'autre.

V-E - Comparison

Introduit dès la version 2.0 du Framework .NET, Comparison permet de comparer deux objets d'un même type, passé comme type générique. Traditionnellement, le résultat de cette comparaison sera représenté par un nombre entier, qui sera égal à zéro si les deux objets sont considérés comme égaux, supérieur à un si le premier objet passé en paramètre a une valeur relative supérieure au second, et inférieur à un dans le cas inverse.

Un exemple de méthode permettant la comparaison et pré-incluse dans le Framework .NET est string.Compare.

```
string.Compare("A", "B");
```

Cette opération renverra -1 car « A » venant avant « B » dans l'alphabet, sa valeur relative lui est considérée comme inférieure.

Depuis le début de cet article, nous cherchons à trier un tableau de chaînes de caractères. Or il existe une méthode toute prête pour effectuer cette opération : la méthode Array.Sort. Celle-ci - parmi ses nombreuses surcharges - accepte en paramètres le tableau que l'on souhaite trier, et un délégué de type Comparison. Ainsi pour trier le contenu de notre tableau de chaînes de caractères, nous pourrions écrire :

```
Array.Sort(data, new Comparison<string>(string.Compare));
```

Nous aurions pu écrire une expression lambda personnalisée pour comparer deux chaînes de caractères, mais ici nous passons directement au délégué le groupe de méthodes string.Compare. Cette syntaxe est rendu possible par le fait que l'une des surcharges de la méthode string.Compare respecte scrupuleusement la signature attendue par le délégué Comparison<string>, à savoir deux chaînes de caractères passées en paramètres, et un nombre entier comme type de retour. Cette syntaxe est donc strictement équivalente à :

```
Array.Sort(data, new Comparison<string>((s1, s2) => string.Compare(s1, s2)));
```

Supposons maintenant que nous devions comparer deux objets de type nombre entier, et que nous devions pour cela implémenter nous-mêmes Comparison. Cela pourrait donner :

```
Comparison<int> intComparison = (i1, i2) =>
                                    {
                                        if (i1 < i2)
                                            return -1;
                                        else if (i1 > i2)
                                            return 1;

                                        return 0;
                                    };
```

Grâce à ce délégué Comparison nous pourrions trier un tableau de nombres entiers de la façon suivante :

```
int[] numbers = new int[] { 42, 15, 4, 8, 23, 16 };
Array.Sort(numbers, intComparison); // Les nombres dans le tableau sont remis dans l'ordre
```

V-F - EventHandler

EventHandler a été introduit dès la version 1.0 du Framework .NET, et dans la 2.0 pour sa version générique.

Nous l'avons vu plus haut, la création d'événements en C# est assujettie à celle d'un délégué qui va encapsuler les méthodes s'y abonnant. Les bonnes pratiques stipulent que le délégué en question doit satisfaire la signature :

```
public delegate void EventHandler(object sender, EventArgs e);
```

Or c'est précisément, comme on le voit, la signature définie par EventHandler, qui est donc un délégué tout indiqué pour servir de base à la création d'événements.

Nous avons vu aussi qu'il était possible d'utiliser à la place d'EventArgs n'importe quel type en héritant. Voici un candidat :

```
public class CustomEventArgs : EventArgs
{
    public bool IsRaisedByChuckNorris { get; set; }
}
```

La version générique d'EventHandler va nous permettre d'utiliser cette classe très facilement pour déclarer un événement :

```
public event EventHandler<CustomEventArgs> OnCustomEvent;
```

Et voici un exemple de méthode qui pourra s'abonner à cet événement :

```
public void OnCustomEventRegisteredMethod(object sender, CustomEventArgs e)
{
    if (e.IsRaisedByChuckNorris)
        ScreamAndRun();
}
```

V-G - Conclusion

Tous ces délégués clef en main contribuent à rendre l'utilisation du Framework .NET encore plus aisée. Malgré tout, il existe encore des cas particuliers où la déclaration d'un délégué « à l'ancienne » est nécessaire, notamment lorsque l'on souhaite utiliser la récursivité (un délégué qui s'appelle lui-même).

VI - .NET 4.0, la covariance et la contravariance

La covariance existait déjà dans la première livraison du Framework .NET ; c'est elle qui permettait d'effectuer ce genre de manipulations :

```
object[] objectArray = new string[0];
```

Ce casting, puisque c'est ainsi qu'on l'appelle, n'était possible que sur des tableaux. Par contre ceci était impossible :

```
Func<object> func = new Func<string>(() => string.Empty);
```

À partir du Framework .NET 4.0, cette opération devient possible sur les délégués comme celui-ci, ainsi que sur les interfaces, mais uniquement lorsque ceux-ci sont génériques, et uniquement avec les types références, pas les types valeurs comme int, double, byte, etc...

Mais d'abord, un rappel s'impose.

VI-A - Le polymorphisme

En .NET, les objets peuvent hériter des attributs d'autres objets. C'est ce qu'on appelle l'héritage.

Sur cette notion vient se greffer celle de polymorphisme. Puisque les objets peuvent hériter les uns des autres selon une hiérarchie, ils peuvent aussi se substituer les uns aux autres, comme ceci :

```
Stream stream = new MemoryStream();
object streamThroughPolymorphism = stream;
```

Ces opérations sont possibles car MemoryStream hérite de Stream, et Stream hérite de object (en C# tout hérite de object). On dit que MemoryStream « est un » Stream et que Stream « est un » object.

C'est grâce au polymorphisme et à la généricité que nous pouvons créer ce genre de collections :

```
IList<Stream> collection = new List<Stream>
                           {
                               new MemoryStream(),
                               new DeflateStream(new MemoryStream(), CompressionMode.Compress),
                               new GZipStream(new MemoryStream(), CompressionMode.Compress),
                               new FileStream(@"C:\", FileMode.Create),
                               new NetworkStream(new Socket(new SocketInformation())),
                               new PrintQueueStream(new PrintQueue(new
 PrintServer("foobar"), "foobar"), "foobar")
                           };
```

Tous les objets que contient cette collection pourront être manipulés comme des Stream, car ils en héritent tous. Mais à moins de les recaster vers leur type d'origine, seuls les membres de la classe Stream seront accessibles.

VI-B - La covariance

Grâce à la covariance, nous allons pouvoir substituer le type de sortie d'un délégué.

En effet, nous avons vu que lorsque nous créons un délégué, nous définissons en fait une signature de méthode, qui permet de désigner quelles méthodes ce délégué va pouvoir encapsuler. Dans cette signature, il n'y a un type de retour, qui peut être void lorsqu'un délégué ne renvoie rien. Si ce type de retour est générique, nous allons pouvoir indiquer que celui-ci est covariant, grâce au mot-clef « out ».

Justement, observons la signature du délégué Func que nous avons utilisé au début de ce chapitre, dans sa version 4.0 :

```
public delegate T Func<out T>();
```

Son type générique T est marqué par notre mot-clef « out », indiquant ainsi que ce type est covariant et sera utilisé comme type de retour, ce qui est d'ailleurs bien le cas ici.

Ainsi lorsqu'on exécute ce délégué, il est possible de récupérer son résultat dans n'importe quel type dont hérite le type que nous lui passons comme type de retour. Par exemple :

```
Stream stream = (new Func<MemoryStream>(() => new MemoryStream()))();
```

Remarquez ici que nous créons un délégué de type Func qui retourne un MemoryStream, et que nous l'exécutons tout de suite (notez bien les parenthèses à la fin de la ligne). Bien que ce délégué renvoie un MemoryStream, nous sommes en mesure, grâce à la covariance, de récupérer son résultat dans une variable de type Stream.

VI-C - La contravariance

Bien entendu, la contravariance permet d'effectuer l'opération exactement inverse. Nous allons pouvoir substituer le type d'un paramètre d'entrée par un autre type qui en hérite.

Comme nous l'avons fait avec Func, observons la signature du délégué générique Action :

```
public delegate void Action<in T>(T param);
```

Ce délégué Action prend un paramètre d'entrée dont le type est générique, et qui est marqué par le mot-clef « in ». Comme vous l'aviez deviné c'est ce mot-clef qui indique que le paramètre d'entrée va être contravariant.

Ainsi lorsque l'on exécute ce délégué, il est possible de lui passer en paramètre n'importe quel type d'objet, du moment que celui-ci hérite du type indiqué comme paramètre d'entrée. Par exemple :

```
(new Action<object>(o => { }))(string.Empty);
```

De la même manière que pour l'exemple précédent, nous déclarons ici un délégué de type Action que nous exécutons dans la foulée. Nous avons indiqué en le déclarant que celui-ci attendait un unique paramètre d'entrée de type object. Pourtant, grâce à la contravariance, lorsque nous l'exécutons nous lui passons un objet de type string.

VI-D - Utilisation simultanée de la covariance et de la contravariance

Dans les deux exemples précédents, nous avons fait la démonstration de la covariance et de la contravariance avec les délégués clef en main, mais nous pouvons bien évidemment utiliser les mots-clefs « in » et « out » sur nos propres déclarations de délégués.

```
public delegate TOut BothCoAndContravariantDelegate<in TIn, out TOut>(TIn param);
```

Nous pouvons d'ailleurs, comme c'est le cas ici, utiliser ces deux mots-clefs sur des paramètres génériques différents. Mais attention ! Un paramètre générique de délégué ou d'interface ne peut pas être à la fois covariant et contravariant.

```
public delegate T BothCoAndContravariantDelegate<in out T>(T param); // Impossible !
```

Enfin, voici un exemple d'utilisation de notre délégué cumulant covariance et contravariance :

```
var method = new BothCoAndContravariantDelegate<object, MemoryStream>(s => new MemoryStream());
```

```
Stream result = method("foobar");
```

Lorsque nous exécutons le délégué que contient la variable « method », nous appliquons à la fois la covariance et la contravariance.

Ainsi, alors que la signature d'origine du délégué indiquait que celui-ci attendait un object en guise de paramètre d'entrée, nous lui passons le string « foobar », ce qui est possible car string hérite de object (string « est un » object).

De la même manière, alors que la signature du délégué indiquait que celui-ci renvoyait un MemoryStream, nous récupérons son résultat dans une variable de type Stream, ce qui est possible car MemoryStream hérite de Stream (MemoryStream « est un » Stream).

Notez enfin que si nous supprimons les mots-clefs « in » et « out » de la déclaration de notre délégué, le programme ne compilera plus.

VII - Conclusion

Nous avons vu que les délégués dans le Framework .NET permettent de rendre votre code plus compact, en simplifiant de nombreuses opérations qui seraient autrement bien plus verbeuses.

Ils constituent aussi un mécanisme puissant, permettant d'appliquer de nombreux patterns pour améliorer la logique de fonctionnement de vos applications. Grâce au principe de l'exécution retardée, vous êtes en mesure de brancher différentes parties de votre code sans que celles-ci soient conscientes de ce qu'elles exécutent.

Cet article se contente d'expliquer l'historique et les bases des délégués, mais leur utilisation est au coeur de nombreux scénarios de développement avancé, par exemple pour tout ce qui touche à la réflexion.

VII-A - Pour aller plus loin

Les délégués sur MSDN :

- **delegate** ;
- **event** ;
- **les méthodes anonymes** ;
- **les expressions lambda** ;
- **Func** ;
- **Action** ;
- **Predicate** ;
- **Converter** ;
- **Comparaison** ;
- **EventHandler** ;
- la **covariance et la contravariance** dans les délégués ;
- les classes **Delegate** et **MulticastDelegate**, dont héritent tous les délégués, et qui permettent d'exécuter ceux-ci par réflexion.

D'autres articles :

- **les expressions lambda pour les débutants** ;
- **la covariance et la contravariance pour les débutants** ;
- **un autre article sur la covariance et la contravariance** (en anglais).

VII-B - Remerciements

Merci à **eusebe19** pour ses corrections et sa relecture attentive, et aux membres de la rédaction .NET pour leurs suggestions.