

## Expression lambda : rappel

Une expression lambda est une fonction anonyme (fonction qui ne porte pas de nom et qui est encodée directement à l'endroit souhaité) qui peut contenir des expressions et des instructions, et qui peut être utilisée pour créer des délégués ou des types d'arborescence d'expression.

## DÉLÉGUÉS

### I. Objectif

Le but d'un délégué est de permettre la déclaration et la codification d'une fonctionnalité tout en différenciant l'implémentation. En effet, une méthode peut effectuer une action qui implique l'appel d'une autre méthode, encore inconnue au moment de la compilation. Un délégué est un élément C# qui permet de référer une méthode. Par conséquent, la définition d'un délégué signale au compilateur quel type de méthode représente le délégué.

### II. Démarche par un exemple

Dans un logiciel, une des tâches est de trier des données sous forme de chaînes de caractères; on souhaite offrir à l'utilisateur le choix parmi une multitude de tris.

#### A. Solution trop longue

```
// classe comprenant tous les tris possibles
public static class SortingAlgorithms
{
    public static void BubbleSort(string[] data)
    {
        ...
    }
    public static void QuickSort(string[] data)
    {
        ...
    }
    // autres méthodes de tri ...
}

// méthode centrale qui permet de trier les données selon le tri choisi
... static void SortArray(string[] data, SortingTypes sortingType)
{
    switch (sortingType)
    {
        case SortingTypes.BubbleSort :
            SortingAlgorithms.BubbleSort (data);
            break;
        case SortingTypes.QuickSort :
            SortingAlgorithms.QuickSort (data);
            break;
    }
}
```

```

    ...
}
}

// énumération
public enum SortingTypes
{
    BubbleSort,
    QuickSort,
    .....
}

class Program
{
    static void Main(string[] args)
    {
        string[] data = new string[] { "iesn", "ulg", "henallux", "uNamur" };
        SortArray(data, SortingTypes.BubbleSort);
    }
}

```

## B. Solution

Handicap : la lisibilité... due à une énumération et un switch trop longs.

Constatons que les méthodes de tri ont le même nombre de paramètres d'entrée (un seul dans notre cas), le même type d'entrées et le même type de retour.

Abstraction par un délégué :

```
public delegate void DelegateSortingMethod (string data[]);
```

Cela correspond plus ou moins à la signature d'une méthode.

Cette déclarative se place soit dans un *namespace* soit dans une classe. Au même titre que les classes, les énumérations ou les structures, les délégués sont des "*first-class citizens*", des membres de plus haut niveau qui créent un nouveau type à part entière dès leur déclarative. Dans le délégué, les méthodes de tri vont être enregistrées : le délégué encapsule une ou plusieurs méthodes.

```

namespace DelegateProgram
{
    public delegate void DelegateSortingMethod(string[] data);

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

        private static void SortArray(string[] data, DelegateSortingMethod sortingMethod)
        {
            sortingMethod(data);
        }
    }
}

```

```
}  
  
}
```

Plus d'énumération!

Instanciation d'un objet `sortingMethod` de type délégué

**DelegateSortingMethod** `sortingMethod` = new **DelegateSortingMethod** (`SortingAlgorithms.BubbleSort`);  
avec un constructeur recevant un argument qui est la référence de la méthode (nom sans `()`), la méthode ne devant pas à ce niveau être exécutée). C'est ce qu'on appelle un groupe de méthodes ou "method group". Ici, `BubbleSort` pourrait avoir plusieurs surcharges, lesquelles seraient incluses d'office.

L'instruction

```
sortingMethod(data);
```

implique l'exécution du tri qui est encapsulé.

La méthode `SortArray` exécute le délégué passé en paramètre en ignorant quelle méthode de tri le délégué encapsule.

On peut référencer plusieurs méthodes dans un délégué appelé multicast dans ce cas.

```
sortingMethod += SortingAlgorithms.QuickSort;
```

Lors de l'exécution du délégué, les méthodes sont exécutées à tour de rôle dans l'ordre de référencement.

On peut déréférencer :

```
sortingMethod -= SortingAlgorithms.QuickSort;
```

On peut aussi simplifier (pas de constructeur) :

```
DelegateSortingMethod sortingMethod = SortingAlgorithms.BubbleSort;
```

ou encore

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

### III. Déclarative

Il s'agit de la réplique de la déclarative des méthodes correspondantes :


```
type_accès delegate type_retour nom_délégué (liste arguments);
```

#### Exemples

- public **delegate** int `MyDelegateAfficher`(char choix);
- private **delegate** string `MyDelegate`();

- protected **delegate** void AlarmEventHandler(object sender, AlarmEventArgs e);  
Par convention, les délégués d'événement figurant dans le .NET Framework possèdent deux paramètres, la source qui a déclenché l'événement et les données de l'événement.

#### IV. Exemples d'application

- Démarrage d'une thread :  
Thread.start (...);  
 appel de la méthode à invoquer
- Bibliothèques génériques  
Exemple : méthode de tri de tableau d'objets  
C'est le client qui décide comment comparer les objets.
- Gestion d'événements : l'émetteur lorsqu'il crée un événement peut ne pas connaître l'objet/la méthode qui va gérer l'événement.  
Les délégués servent d'intermédiaires : ils informent le runtime.net des méthodes qui gèrent les événements (EventHandler).

#### V. Classe déléguée

Définir un délégué, c'est définir une nouvelle classe qui contient une référence à une méthode mais elle s'écrit sous forme de signature à la différence des autres classes.

Le compilateur est averti de la classe déléguée et utilise sa syntaxe de délégué pour épargner au développeur les détails de l'opération de cette classe.

Un délégué est par conséquent l'équivalent d'un rappel de fonction ou d'un pointeur de fonction de type sécurisé comme expliqué ultérieurement.

Une déclaration **delegate** suffit pour définir une classe délégué. La déclaration fournit la signature du délégué et le CLR assure l'implémentation.

Les délégués sont implémentés comme des classes dérivées de la classe

**System.Multidelegate**, elle-même dérivée de **System.Delegate** dans le .Net Framework; leur type est **sealed** (non dérivables). En effet, il est impossible de dériver des classes personnalisées de la classe Delegate.

#### VI. Instanciation

Une instance du délégué peut être réalisée de manière à l'utiliser pour stocker des détails d'une méthode particulière. Cette instance n'est pas appelée un objet mais aussi délégué. C'est le contexte qui indique s'il s'agit de la définition du délégué ou d'une instance du délégué.

Exemple

```
private delegate string MyDelegate();
static void Main()
{
    int x = 40;
    MyDelegate firstStringMethod = new MyDelegate (x.ToString);
    Console.WriteLine("String is {0}", firstStringMethod.Invoke());
}
```

**Une instance de délégué donné peut se référer à toute méthode, statique ou d'instance, sur n'importe quel objet de n'importe quel type, à condition que la signature de la méthode corresponde à la signature du délégué.**

Les délégués sont de **type sécurisé** dans la mesure où ils garantissent que la signature de la méthode appelée est correcte.

## VII. Autres exemples

### A. Simplification des écritures

```
private delegate string GetAString();
static void Main()
{
    int x = 40;
    GetAString firstStringMethod = x.ToString; // "delegate inference"
    Console.WriteLine("String is {0}", firstStringMethod);
}
```

### B. Tableaux

```
public static class OperationsBasiques
{
    public static int add(int a, int b)
    {
        return a + b;
    }
    public static int produit (int a, int b)
    {
        return a * b;
    }
}

public class Test
{
    ... static void calcul (Operation action, int a, int b, int pas)
    {
        Console.WriteLine("a = {0}, b = {1}, resultat = {2}", a, b, action(a,b)/(double)pas );
    }
}
```

```

static void Main()
{
    Operation[] operations ={ OperationsBasiques.add, OperationsBasiques.produit };
    for (int i=0; i < operations.Length; i++)
    {
        calcul(operations[i], 2,3,4);
    }
}
} // sachant que Operation est un délégué

```

### C. Tri

<pre> class Enfant {     public string Nom { get; private set; }     public int Age { get; private set; }      public Enfant(string nom, int age)     {         Nom = nom;         Age = age;     }     public override string ToString()     {         return Nom + " est âgé de " + Age + " ans";     }     public static bool CompareAge         (Enfant e1, Enfant e2)     {         return e1.Age &gt; e2.Age;         // tri décroissant     } } </pre>	<pre> class Program {     static void Main()     {         Enfant[] tabEnfants ={             new Enfant("Zorro", 5),             new Enfant("Batman", 10),             ...         };         BubbleSorter.Sort(tabEnfants,             Enfant.CompareAge);         foreach (var enfant in tabEnfants)             Console.WriteLine(enfant);     } } </pre>
---	---

## VIII. Délégués existants

Certains délégués permettent de ne pas écrire la déclarative.

Citons Action, Func, Predicate, Converter, Comparison ainsi qu'EventHandler.

### A. Délégués génériques Action<T> et Func<T>

Plutôt que de définir un nouveau délégué dans le cas où il ne renvoie rien, on peut utiliser le délégué générique Action<T> qui existe sous diverses formes (pas de paramètre, un paramètre, deux paramètres, jusqu'à 16 paramètres de type différent).

#### Exemple

Action<in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8> pour 8 paramètres.

De même, le délégué générique Func<T> peut être utilisé pour des méthodes avec un type de retour ; quant aux paramètres, il peut ne pas y en avoir ou il peut y en avoir jusque 16 différents, le type de retour est toujours le dernier de la liste.

### Exemples

Func<out TResult> : délégué avec un type de retour et pas de paramètre.

Func<in T1,in T2, in T3, in T4, out TResult> : 4 paramètres et un résultat.

### Applications

- L'exemple du point VII.B. peut s'écrire aussi :

```
Func <int, int, int>[] operations = { OperationsBasiques.add, OperationsBasiques.produit };

static void calcul (Func<int, int, int> calculelem, int a, int b, int pas){
    Console.WriteLine ("a = {0}, b = {1}, resultat = {2}", a, b, calculelem (a, b)/( double)pas);
}
```

- Dans l'exemple du tri :

```
public delegate void DelegateSortingMethod(string[] data);
private static void SortArray(string[] data, DelegateSortingMethod sortingMethod) { ... }
```

nous pourrions écrire :

```
private static void SortArray(string[] data, Action <string[]> sortingMethod) { ... }
```

- Via Action, nous pouvons vider une collection de chaînes de caractères :

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

List<string> lstBidon = new List<string> { "riri", "avc" };
actionClear(lstBidon);
```

## **B. Délégué générique Predicate**

Il s'agit de Func qui renvoie du booléen.

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

System.Console.WriteLine(predicate(5));
```

### C. Délégué générique Converter

But : convertir un élément d'un certain type en un autre type. Les deux arguments sont dès lors le type source et le type destination.

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

### D. Délégué générique Comparison

But : comparer deux éléments de même type. Le résultat sera présenté sous forme d'entier (0 en cas d'égalité, négatif ou positif suivant que le 1er paramètre de comparaison est inférieur au 2ième ou supérieur).

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



## IX. Délégués Multicast (Chaînage de délégués)

Les délégués sont de type **SingleCast** ou **MultiCast**.

```
static void calcul (Func <int, int, int> function, int a, int b)
{
    function(a,b);
}
static int add (int a, int b)
{
    Console.WriteLine("{0} + {1} = {2}", a,b,a+b);
    return a +b ;
}
static int product (int a, int b)
{
    Console.WriteLine("{0} * {1} = {2}", a,b,a*b);
    return a *b ;
}

static void Main()
{
    Func<int, int, int> operations = OperationsBasiques.add;
    operations += OperationsBasiques.product;
    calcul(operations, 2,3,4);
}

static void Main(){
    Func<int, int, int> operations = OperationsBasiques.add;
    operations += OperationsBasiques.product;
    Delegate[] delegates = operations.GetInvocationList();
    foreach (Func <int, int, int> fn in delegates){
        calcul (fn,a,b);
    }
}
```

Si une méthode invoquée lève une exception, le processus qui consiste à passer aux méthodes suivantes sera stoppé.

Pour éviter cet arrêt éventuel , on créera un tableau de délégués et on lèvera aussi une exception :

```
static void Main(){
    Action twoAction = One;
    twoAction += Two;
    Delegate[] delegates = twoAction.GetInvocationList();
    foreach (Action action in delegates){
        try {
            action();
        } catch (Exception){
            Console.WriteLine("Exception caught");
        }
    }
}
```

## X. Méthodes anonymes

Il s'agit de déclarer directement le code que devra référencer un délégué, sans avoir à créer une méthode pour le contenir. Création "in line" de la méthode référencée par le délégué. Elle ne porte pas de nom.

En supposant que la déclarative existe bien sous la forme

```
public delegate void DelegateSortingMethod(string[] data);
```

### Exemple1

```
DelegateSortingMethod sortingMethod = new DelegateSortingMethod ( delegate(string[] data)
                                                                    { // Tri ... } );
```

Ou encore :

```
DelegateSortingMethod sortingMethod = delegate(string[] data) { // Tri ... } ;
```

Ou :

```
SortArray (data, delegate(string[]data) { //Tri... } );
```

### Exemple2

Au démarrage d'un thread, le code exécuté par le thread est placé directement sans créer de méthode supplémentaire pour le délégué.

```
Using System.Threading;
void StartThread()
{
    Thread MyThread = new Thread (
        delegate()
        {
            System.Console.Write("Hello, ");
            System.Console.WriteLine("World!");
        }
    );

    MyThread.Start();
}
```

## XI. Expressions LAMBDA

Ce type d'expression est privilégié pour remplacer une méthode anonyme.

En effet, l'expression

```
DelegateSortingMethod sortingMethod = delegate(string[] data) { // tri };
```

peut encore s'écrire autrement :

```
DelegateSorting sortingMethod = new DelegateSortingMethod ( (string[] data) => { // TRI... } );
```

L'opérateur => se lit "conduit à ". Le mot delegate a disparu.

Le compilateur étant capable d'inférer automatiquement le type des paramètres d'entrée et le type de retour à partir d'un délégué, on peut écrire :

```
DelegateSorting sortingMethod = new DelegateSortingMethod ( (data) => { // tri... } );
```

```
private delegate int DelegateCarre(int i);

static void Main(string[] args)
{
    DelegateCarre myDelegate = x => x * x;
    int j = myDelegate(5);                //j = 25
}
```

Mais il existe un cas où une méthode anonyme fournit des fonctionnalités introuvables dans les expressions lambda. Une méthode anonyme permet d'omettre la liste de paramètres. Cela signifie qu'une méthode anonyme peut être convertie en délégués avec diverses signatures. Ceci est impossible avec les expressions lambda.

## XII. Délégués et méthodes génériques

### Exemple

T1, T2 : paramètres génériques

Accumulate : méthode avec deux paramètres :

- le premier étant une source de type collection IEnumerable<T1>
- le second, un délégué générique Func

```
public class Algorithm
```

```
{
    // retour de la méthode du même type que T2
    public static T2 Accumulate<T1, T2> (IEnumerable<T1> source, Func<T1, T2, T2> action)
    {
        T2 sum = default(T2);
        // sum initialisé à la valeur par défaut selon le type de T2
        foreach (T1 item in source)
        {
            sum = action(item, sum);
        }
        return sum;
    }
}
```

Diagramme d'annotation :

- ↑ retour de la méthode du même type que T2 (pointe vers T2 dans la signature)
- collection (pointe vers IEnumerable<T1> source)
- délégué (pointe vers Func<T1, T2, T2> action)
- paramètres génériques (pointe vers T1 et T2 dans la signature)

Appel :

```
var accounts = new List<Account>()
{
    new Account("Christian", 1500),
    new Account("Stephanie", 2200),
    new Account("Angela", 1800)
};
decimal amount = Algorithm.Accumulate<Account, decimal>
(accounts, (item, sum) => sum += item.Balance);
```

où

```
public class Account
{
    public string Name { get; set; }
    public decimal Balance { get; set; }
    public Account(string name, decimal balance)
    {
        this.Name = name;
        this.Balance = balance;
    }
}
```

### XIII. Covariance/contravariance

Les méthodes ne doivent pas nécessairement correspondre à la signature du délégué !

#### A. Covariance

```
class Mammals{}
class Dogs : Mammals{}
class Program
{
    public delegate Mammals HandlerMethod();
    public static Mammals MammalsHandler()
    {
        return null;
    }
    public static Dogs DogsHandler()
    {
        return null;
    }
    static void Test()
    {
        HandlerMethod handlerMammals = MammalsHandler;
        // Covariance
        HandlerMethod handlerDogs = DogsHandler;
    }
}
```

#### B. Contravariance

Les délégués peuvent être utilisés avec les méthodes ayant des paramètres d'un type qui sont les types de base du type de paramètre de la signature du délégué.

Plutôt que d'écrire deux gestionnaires d'événements séparés (le premier lié à l'enfoncement d'une touche clavier, le deuxième lié au click de la souris), on n'écrit qu'un seul gestionnaire qui accepte un paramètre EventArgs.

```
private void MultiHandler(object sender, System.EventArgs e)
{
    label1.Text = System.DateTime.Now.ToString();
}

public Form1()
{
    InitializeComponent();
    // utilisation de la méthode avec un EventArgs quoique l'événement attend un KeyEventArgs
    this.button1.KeyDown += this.MultiHandler;
    // utilisation de la méthode avec un EventArgs quoique l'événement attend un MouseEventArgs
    this.button1.MouseClick += this.MultiHandler;
}
```

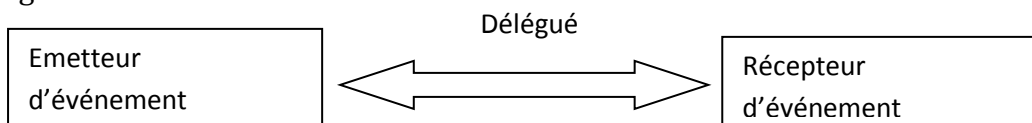
## ÉVÉNEMENTS

A chaque clic sur un bouton dans une application Windows, un événement est déclenché. Dans l'application qui comprend ce bouton, une action doit se produire; il faut en informer les objets abonnés à cet événement.

### I. Concepts

Dès lors, un événement est un message transmis par un objet pour signaler l'occurrence d'une action. L'action peut être provoquée par une interaction utilisateur ou peut être déclenchée par une autre logique de programme. L'objet qui déclenche l'événement est appelé émetteur d'événement. L'objet qui capture l'événement et y répond est appelé récepteur d'événements. On appelle notification d'événements le message reçu - par exemple, par l'application dans la fenêtre où on a cliqué. Dans ce cas, le responsable de la notification est Windows; on l'appelle générateur d'événements. Ce pourrait être le .Net Framework.

Lors de la transmission d'un événement, la classe de l'émetteur d'événement ne connaît pas l'objet ou la méthode qui recevra (gérera) les événements qu'elle déclenche. Un intermédiaire s'avère nécessaire entre l'émetteur et le récepteur. Pour ce faire, le .NET Framework utilise le délégué.



Les événements sont des formes particulières de délégués; en effet, il s'agit d'une forme spécialisée de délégués **multicast**. Dès lors, ils peuvent contenir des références à plusieurs méthodes de gestion des événements. Les délégués assurent une souplesse et un contrôle précis lors de cette gestion des événements. Un délégué joue le rôle de répartiteur d'événement pour la classe qui déclenche l'événement en tenant à jour une liste des récepteurs d'événements inscrits pour l'événement.

Soit une classe TrainGare représentant une gare ferroviaire qui aura la responsabilité d'informer de l'arrivée d'un train, grâce à un événement TrainArrival.

```

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

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

**La création d'un événement** est soumise à la **création d'un délégué** ayant une **signature précise** requise par le .Net Framework :

- un retour de type void. Le gestionnaire appelle une méthode; dès lors, il ne renvoie rien sauf exceptionnellement;
- comme paramètres :

- une référence à l'objet qui a généré l'événement (déterminera quelle source est responsable de l'événement s'il existe plusieurs sources possibles de l'événement - exemple : plusieurs boutons)
- une référence soit à la classe .Net **System.EventArgs** (classe de base générique pour toute notification d'événement) soit à une classe dérivée de celle-ci. Ce deuxième paramètre va encapsuler les arguments de l'évènement.

La **déclaration de l'événement** se fait avec le mot-clé **event** en indiquant quel délégué sera chargé d'encapsuler les méthodes qui vont s'abonner à cet événement. Ce délégué servant de gestionnaire d'événement est nommé traditionnellement du même nom que l'événement lui-même en le suffixant avec **EventHandler**.

Abonnons la classe Ecouteur à l'évènement TrainArrival d'un objet trainGare:

```
class Ecouteur
{
    static void Main(string[] args)
    {
        TrainGare trainGare = new TrainGare ();
        trainGare.TrainArrival += OnTrainArrival;    // mécanisme identique au délégué
    }

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

Plusieurs méthodes pourraient être encapsulées ainsi que des méthodes anonymes. Le fait de préfixer le nom de la méthode par "On" et de lui donner le même nom que l'événement est une convention de nommage.

Pour que la méthode *OnTrainArrival* soit exécutée, la classe TrainGare doit exécuter (lever - invoquer) l'événement c'est-à-dire 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, la propriété statique EventArgs.Empty pour indiquer qu'il n'y en a aucun).

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

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

Si aucune méthode n'était abonnée à l'événement, celui-ci serait null, et tenter de le lever provoquerait une exception!

```

TrainGare trainGare = new TrainGare();
trainGare.TrainArrival += OnTrainArrival;
trainGare.RaiseTheEvent();

```

## II. Responsabilité de la levée de l'évènement

Il est nécessaire de bien analyser les déclaratives des événements et méthodes associées. Dans notre cas, l'évènement était public; tout le programme avait la possibilité de s'abonner au délégué et de l'exécuter.

La méthode RaiseEvent qui va lever l'évènement était aussi publique; si nous souhaitons que seule la classe TrainGare soit en mesure de l'appeler et donc de lever l'évènement, la méthode devait être de type private.

Les événements ont deux accesseurs **add** et **remove** de la même manière que les propriétés disposent de **get** et **set**. Si l'un est écrit, l'autre doit l'être aussi, même s'il est vide. Grâce à ces accesseurs, il est possible d'effectuer des actions particulières lorsqu'une méthode s'abonne ou se désabonne d'un événement.

Mais l'utilisation de ces accesseurs nécessite de modifier la classe TrainGare.

La méthode qui souhaite s'abonner à l'évènement doit être encapsulée dans le délégué, en utilisant le mot-clé **value**.

```

public class TrainGare
{
    private TrainArrivalEventHandler trainArrival;

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

        remove
        {
            Console.WriteLine("Impossible de se désabonner de l'évènement.");
            // aucune méthode n'est désencapsulée dans l'exemple
        }
    }

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

```



Plus important, une instance du délégué `TrainArrivalEventHandler` a dû être créée dans le champ privé `trainArrival`; cette instance est exécutée 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 implicite jusqu'à présent.

Cette obligation est due aux accesseurs grâce auxquels existe la possibilité d'encapsuler les méthodes souhaitant s'abonner à l'événement dans plusieurs délégués différents.

Dans l'exemple ci-dessous, l'application est conçue pour tourner sans interruption pendant plusieurs mois.

Nous encapsulons les méthodes dans deux délégués en fonction de la date courante. En décembre, les méthodes seront encapsulées dans le délégué `trainArrivalDecember`; sinon elles le seront dans le délégué `trainArrivalOtherMonths`.

La méthode `RaiseTheEvent` prend en paramètre un booléen lui indiquant si elle doit ou non exécuter le délégué correspondant au mois de décembre.

```
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);
    }
}
```

### III. Evènements et interfaces

Pour rappel, une interface permet d'abstraire les signatures de certains éléments d'une classe. Abstraire un délégué est impossible. Dès lors, il faut pratiquer comme dans l'exemple ci-dessous.

```
public interface IDrawingObject
{
    event EventHandler ShapeChanged;
}
public class MyEventArgs : EventArgs {...}
public class Shape : IDrawingObject
{
    event EventHandler ShapeChanged;
    void ChangeShape()
    {
        // Do something before the event...
        OnShapeChanged(new MyEventArgs(...));
        // or do something after the event.
    }
    protected virtual void OnShapeChanged(MyEventArgs e)
    {
        if(ShapeChanged != null)
        {
            ShapeChanged(this, e);
        }
    }
}
```

## IV. Démarches via une application

Pour qu'une classe déclenche un événement, trois éléments sont nécessaires

- une classe qui fournit les données d'événement ;
- un délégué d'événement ;
- la classe qui déclenche l'événement.

### 1. Définition d'une classe pour fournir les données d'événement

Par convention dans le .NET Framework, lorsqu'un événement est déclenché, il passe les données d'événement à ses gestionnaires d'événements. Les données d'événement sont fournies par la classe *System.EventArgs* ou par une classe qui en est dérivée.

Soit un événement n'a pas de données personnalisées ; toutes les informations requises par les gestionnaires d'événements ont été fournies suite au déclenchement de l'événement. Dans ce cas, l'événement peut passer un objet *EventArgs* à ses gestionnaires. La classe *EventArgs* n'a qu'un seul membre, *Empty*. Il peut être utilisé pour instancier une nouvelle classe *EventArgs*.

Soit un événement a des données personnalisées, il peut passer une instance d'une classe dérivée de *EventArgs* aux gestionnaires d'événements. En fonction des données précises que l'événement passe aux gestionnaires, il se peut qu'une classe de données d'événement existante dans le .NET Framework puisse être utilisée. Par exemple, si le gestionnaire d'événements autorise l'annulation de l'action associée à l'événement, la classe *CancelEventArgs* peut être utilisée.

Lorsque des données personnalisées doivent être transmises aux gestionnaires d'événements et qu'aucune classe existante n'est disponible, une classe de données d'événement doit être écrite et dériver de *System.EventArgs*.

```
public class AlarmEventArgs : EventArgs
{
    private DateTime alarmTime;
    // indique quand l'alarme se déclenche
    private bool snoozeOn = true;
    // indique si l'alarme doit se déclencher à nouveau après un intervalle
    // donné ou si les alarmes ultérieures doivent être annulées

    public AlarmEventArgs(DateTime time)
    {
        this.alarmTime = time;
    }

    public DateTime Time
    {
        get { return this.alarmTime; }
    }

    public bool Snooze
    {
        get { return this.snoozeOn; }
        set { this.snoozeOn = value; }
    }
}
```

## 2. Définition d'un délégué pour l'événement

Un délégué d'événement est utilisé pour définir la signature de l'événement. Par convention, dans le .NET Framework, les événements ont la signature `EventHandler(sender, e)`, où `sender` est un *Object* qui fournit une référence à la classe ou la structure qui a déclenché l'événement, et `e` est un objet *EventArgs* ou un objet dérivé de *EventArgs* qui fournit les données d'événement.

Généralement, la définition de délégué prend alors la forme `EventHandler(sender, e)`.

Si on utilise une classe de données d'événement déjà définie dans la bibliothèque de classes .NET Framework ou dans une bibliothèque tierce, il est possible qu'un délégué d'événement correspondant soit aussi défini dans cette bibliothèque. Par exemple, le délégué *EventHandler* peut être utilisé avec la classe *EventArgs*. De même, le délégué *CancelEventHandler* peut être utilisé avec la classe *CancelEventArgs*.

Si une classe de données d'événement est personnalisée, un délégué personnalisé peut être écrit pour définir la signature d'événement, ou utiliser le délégué *Action<T1, T2>* générique.

Soit un délégué d'événement nommé `AlarmEventHandler`.

```
public delegate void AlarmEventHandler(object sender, AlarmEventArgs e);
```

## 3. Définition d'une classe pour déclencher l'événement

La classe qui déclenche l'événement doit fournir la déclaration de l'événement et définir une méthode qui déclenche l'événement. De plus, elle doit fournir une certaine logique pour déclencher l'événement dans une propriété ou une méthode de classe.

```
public class Alarm
{
    private DateTime alarmTime;
    private int interval = 10;

    public event AlarmEventHandler AlarmEvent;

    public Alarm(DateTime time) : this(time, 10)
    {
    }

    public Alarm(DateTime time, int interval)
    {
        this.alarmTime = time;
        this.interval = interval;
    }

    // méthode contenant la logique qui va déclencher l'évènement
    public void Set()
    {
        while (true) {
            System.Threading.Thread.Sleep(2000);
            DateTime currentTime = DateTime.Now;
```

```
if (currentTime.Hour == alarmTime.Hour &&
    currentTime.Minute == alarmTime.Minute)
{
    // instanciation d'un objet AlarmEventArgs avec l'heure à laquelle
    // l'alarme s'est déclenchée
    AlarmEventArgs args = new AlarmEventArgs(currentTime);
    OnAlarmEvent(args);
    if (args.Snooze == false)
        // plus d'évènement d'alarme à déclencher
        return;
    else
        // incrémentation de l'heure de l'alarme
        this.alarmTime = this.alarmTime.AddMinutes(this.interval);
}
}
}

// méthode chargée du déclenchement de l'évènement
protected void OnAlarmEvent(AlarmEventArgs e)
{
    AlarmEvent(this, e);
}
}
```