



Concepteur Développeur en Informatique

Développer des composants d'interface

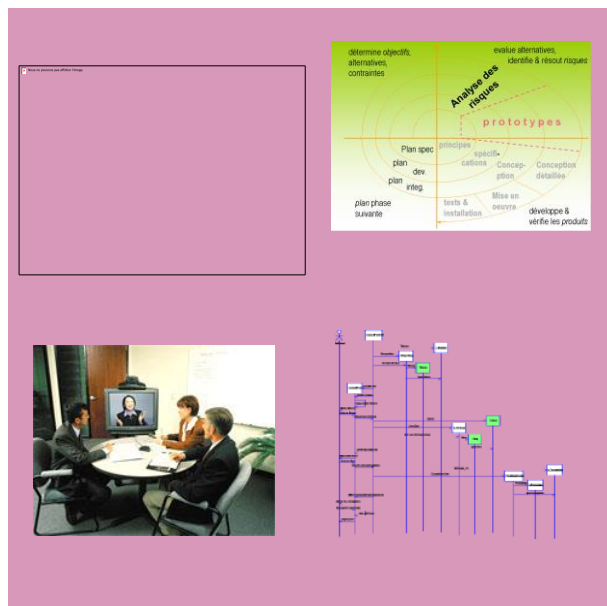
Programmation Orientée Objet – Evénements

Accueil

Apprentissage

PAE

Evaluation



Localisation : U03-E03-S02

SOMMAIRE

1. Introduction.....	2
2. Classes de délégation	3
2.1. Principe du délégué	3
2.1.1. Quel est l'intérêt d'utiliser un délégué ?	5
3. Classe d'événement et délégué.....	6
3.1. Règles de nommage et éléments sollicités.....	6
3.2. Données d'événement	7
3.3. Exemple du changement de nom d'une personne	9
4. Déclaration d'un événement avec un délégué d'événement spécialisé.....	12
4.1. Au niveau de la classe Emetteur	12
4.2. Au niveau de la classe Récepteur	12

1. Introduction

Ce dernier document conclue la première période d'apprentissage des principes de la programmation orientée objet en mettant en œuvre deux mécanismes bien connus et associés :

- La délégation de méthode
- Les événements

La délégation de méthode va nous permettre de passer une méthode en paramètre à une autre méthode. Mais pour quoi faire ?

L'un des exemples courants de mise en œuvre des délégués qui permet d'en comprendre l'intérêt est le traitement des événements.

Jusqu'à ce jour, les programmes que nous avons créés ont exécuté des instructions les unes à la suite des autres, dans l'ordre déterminé au moment de la compilation du programme.

Nous n'avons pas vu encore la réalisation des interfaces Windows mais il vous sera toutefois possible de comprendre le principe de la délégation au travers de la gestion des événements liés à une interface de ce type. Imaginons donc une application dotée d'une interface riche de type Windows.

Celle-ci est dotée d'un bouton. Lorsque l'utilisateur clique sur le bouton, cela suscite une réaction du système sous la forme d'un événement.

Les concepteurs des objets de type bouton (contrôle de la classe Winforms.Button) ont donc prévu un comportement précis dans le cas du click de souris sur un bouton.

Par contre ceux-ci ne peuvent évidemment pas prévoir ce que l'application cliente devra réaliser lorsque l'événement du fait d'un click surviendra.

Ils ne sont même pas en mesure de déterminer si l'application cliente devra ou non tenir compte de l'événement.

Considérez maintenant un programme utilisateur implémentant une instance de la classe Button. Lorsque l'utilisateur va cliquer sur le bouton à l'aide de la souris, il va fournir au programme via le système d'exploitation, de nouvelles informations.

Lorsque l'application est informée de l'évènement qui s'est produit, celle-ci doit ou non traiter cet évènement. Il faut donc que l'utilisateur ait défini une méthode permettant de le traiter mais il faut également que l'utilisateur ait fourni au programme le moyen de savoir vers quelle méthode celui-ci doit se tourner lorsque l'évènement est déclenché.

C'est là l'intérêt des délégués : l'utilisateur définit sa méthode et indique, via un délégué, quelle méthode doit être appelée lors de l'évènement en question.

3 principes sont mis en œuvre dans la transaction précédente :

- Le principe d'évènement
- Le principe de délégué
- L'association entre évènement et délégué

2.1. Principe du délégué

Un délégué est un intermédiaire qui permettra de **faire passer la référence de la méthode à invoquer dans un contexte spécifique**. Un délégué permet d'établir un contrat entre deux objets.

Les délégués sont très utilisés dans l'environnement .NET : le mécanisme de gestion des événements est directement basé sur les délégués. Nous verrons qu'il y a d'autres mécanismes qui y ont recours.

La notion de délégué pourrait être comparée à celle de l'interface puisqu'une **interface** est aussi un contrat entre classes. Toutefois un **délégué** n'établit son **contrat** qu'avec une **méthode** et non une classe.

Le contrat du délégué est établi lors de la déclaration de celui-ci.

La signature de la méthode doit être alors précisée complètement : son identificateur, son type, le nombre et le type de ses arguments. Un délégué est donc un **modèle** pour la méthode qui sera réellement exécutée.

Un délégué est un type particulier qui encapsule une méthode. Tous les délégués sont déclarés à partir d'un mot clé spécial **delegate**.

Commençons par un exemple simple.

Nous créons un délégué de méthode vide sans argument.

Nous créons ensuite une variable du type du délégué nouvellement créé à laquelle nous assignons une des méthodes existantes conformes à la signature du délégué. Cette variable stockera des références à des méthodes. Il s'agit ici d'un exemple sans autre intérêt que de voir comment déclarer un délégué et l'utiliser dans un programme. Ici, le délégué et les méthodes sont tous connus et présents au sein d'un même formulaire...

La déclaration du délégué s'appuie toujours sur le type particulier **delegate**

Pour déclarer notre délégué nous utilisons donc **delegate** puis nous précisons le type du délégué. La déclaration d'un délégué s'apparente à une signature de méthode. Nous devons préciser le type de la méthode (void, int, string, ..) et ses arguments.

Dans ce premier exemple, le type délégué sera une méthode vide sans arguments.

```
// Déclaration du type Delegate  
  
delegate void TypeDelegate();
```

Utilisation du délégué. Nous allons recourir à une variable du type délégué qui contiendra des méthodes, concrètes cette fois-ci, respectant la définition du type.

Déclaration d'une variable du type délégué.

Cette déclaration se fait de manière analogue à toute autre déclaration de variable.

```
// Déclaration d'une variable du type Délégué  
  
private TypeDelegate appelMethodes = null;
```

Nous stockons dans cette variable des méthodes concrètes passées au constructeur du type délégué.

Nous voyons au travers de l'IntelliSense que le constructeur attend une méthode vide sans argument :

```
private void btnAppelMethodeA_Click(object sender, EventArgs e)
{
    appelMethodes += new TypeDelegate();
}
private void MethodeA()
```

TypeDelegate.TypeDelegate(void () target)

Cela correspond au type du délégué. Ajoutons une méthode à notre variable.

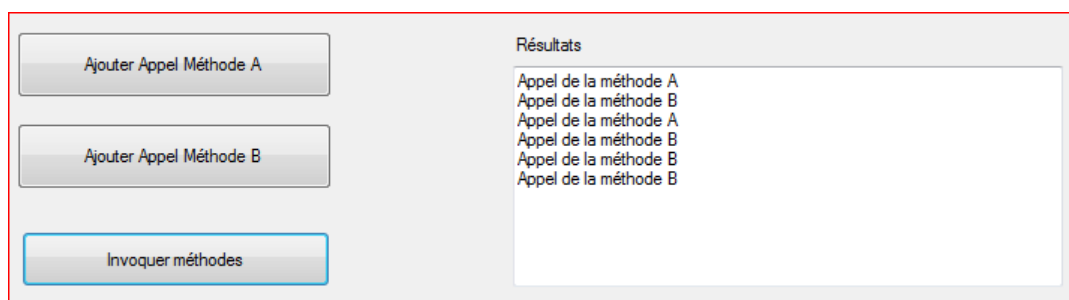
```
private void btnAppelMethodeA_Click(object sender, EventArgs e)
{
    appelMethodes += new TypeDelegate(MethodeA);
}
```

Invocation des méthodes stockées dans notre variable

```
private void btnInvoquerMethodes_Click(object sender, EventArgs e)
{
    appelMethodes();
}
```

Dans l'exemple présenté au complet ci-dessous, le formulaire permet de stocker des appels de méthodes de type A ou B qui mettent à jour une boîte de texte... Exemple d'un grand intérêt !

J'ai cliqué 2 fois sur le bouton Ajouter appel méthode A et 4 fois sur le bouton Appel méthode B.



```
private void MethodeA()
{
    txtAppels.Text += "Appel de la méthode A \r\n";
}
private void MethodeB()
{
    txtAppels.Text += "Appel de la méthode B \r\n";
}
private void btnInvoquerMethodes_Click(object sender, EventArgs e)
{
    txtAppels.Clear();
    appelMethodes();
}
private void btnAppelMethodeB_Click(object sender, EventArgs e)
{
    appelMethodes += new TypeDelegate(MethodeB);
}
```

2.1.1. Quel est l'intérêt d'utiliser un délégué ?

L'intérêt des délégués n'est pas nécessairement perceptible de suite.

Ils répondent cependant à un problème récurrent en programmation :

Fournir un comportement via une méthode à un objet qui l'exécutera par la suite.
L'objet demandera l'exécution d'instructions qui lui sont pour le moment inconnues.
L'exécution peut être demandée de manière synchrone ou asynchrone.

Prenons l'exemple d'une application qui utilise un objet chargé de gérer des communications via le réseau. Elle va utiliser un délégué pour signaler qu'une donnée vient d'arriver et qu'elle est disponible.

Ce mécanisme que les délégués utilisent est de type « rappelle-moi plus tard ».
Ce système de rappel, callback en Anglais, permet de gérer aisément les traitements asynchrones mais pas seulement.

Dans cet exemple la classe qui doit gérer la communication ne connaît rien de l'application qui l'utilisera.

Le code de cette classe est indépendant et doit être **réutilisable** dans d'autres situations **sans aucune modification**.

De manière générale, l'utilisation des délégués permet de créer des classes, des contrôles ou composants utilisateurs plus indépendants les uns des autres (dit à couplage faible) ce qui facilite la réutilisation du code et tout ce qui en découle :

- maintenance,
- évolution
- publication.

Nous mettons en œuvre dans notre exemple la notion de délégué pour traiter des états particuliers d'un objet qui doivent être portés à la connaissance de l'application cliente qui manipule ce même objet. Il s'agit ici du traitement d'un événement (changement d'état).

Lors de la création d'un événement, l'émetteur ne peut pas connaître l'objet / la méthode qui va gérer cet événement.

Les délégués permettent donc de prendre en charge ce rôle d'intermédiaire.

Ils offrent aux développeurs une méthode de dialogue entre objets, en permettant de fournir en paramètre une fonction de même signature pour déléguer l'exécution d'un morceau de code.

Nous aurons donc toujours soin dans une application de faire en sorte que nos objets métiers ne soient jamais couplés fortement aux objets d'interface.

En règle générale nous rechercherons toujours un couplage faible (faible dépendance) entre les objets des différentes couches.

Si le couplage entre ma classe Salarie et l'application Windows qui l'emploie est fort, je risque de rendre mon objet métier uniquement utilisable dans ce contexte et devoir lui faire subir des modifications importantes pour l'utiliser dans une application Web.

Un délégué permet de cibler la référence d'une méthode.
Il permet de passer une méthode en paramètre à une autre méthode.

Lors de la conception de l'objet, le délégué représente une méthode abstraite. La méthode concrète, celle qui sera réellement exécutée, doit uniquement respecter la signature du délégué (même type, même arguments).

3. Classe d'événement et délégué

Un événement est un message transmis par un objet pour signaler l'occurrence d'une action. Les événements sont associés à des changements d'état d'un objet.

L'événement peut être provoqué par une interaction utilisateur, telle qu'un clic de souris, ou peut être déclenché par une autre logique de programme, comme le changement du nom de la personne comme proposé ci-après.

L'objet qui déclenche l'événement est appelé **émetteur d'événement**.

L'objet qui capture l'événement est appelé **récepteur d'événements**.

Nous allons dans l'exemple qui suit créer un exemple simple d'événement.

Nous avons défini une entité Personne.
Ce type Personne possède un champ nom.

Nous souhaitons mettre en place un événement qui se déclenchera lors du changement de nom de la personne.

Cet événement sera donc associé à un objet (instance) de type Personne.

3.1. Règles de nommage et éléments sollicités

Il existe des règles de nommage des éléments mis à contribution lors de la mise en place de communication par événements.

Au niveau de la **classe Emetteur (celle qui déclenche)**

Tout d'abord le nom de l'événement : **EventName**. Dans notre exemple ici, **ChangementNom**.

Lui seront associés :

- Un type délégué qui sera nommé **EventNameEventHandler**, donc ici **ChangementNomEventHandler**.
- L'événement, ici **ChangementNom**, sera défini du type du délégué
- Une méthode qui déclenche l'événement nommée **OnEventName** donc ici **OnChangementNom**

Si des données doivent être transportées avec l'événement, elles seront définies d'un type dérivant de la classe **EventArgs**. Ce type devra être nommé en respectant la forme **EventNameEventArgs**. Donc ici, si l'événement **ChangementNom** doit transporter des infos, un type **ChangementNomEventArgs** sera créé.

La classe Récepteur

Elle devra s'abonner à l'événement si celui-ci l'intéresse.

Elle fournira à l'émetteur la méthode (gestionnaire d'événement) qui devra s'exécuter.

Le gestionnaire d'événement prend par défaut un nom composé de **NomObjet_NomEvenement** de même signature que le type du délégué.

Ainsi, pour un objet **salarie** et un événement **changementNom**, son nom par défaut serait `salarie_changementNom`

3.2. Données d'événement

Les événements peuvent transporter des informations.

Le premier argument de l'événement est toujours l'émetteur.

Le deuxième argument représente les données transportées à l'attention de l'application cliente qui interceptera l'événement et traitera ces données.

Les arguments de l'événement, qui représentent les informations transmises sont toujours d'un type dérivé de la classe de base **EventArgs**.

4. Premier exemple

Voyons un premier exemple concret pour faciliter la compréhension de ces mécanismes.

Nous allons ici mettre en œuvre un événement sans transport d'information qui surviendra lorsque la propriété `Prenom` de l'objet `Personne` sera modifiée.

4.1. Au niveau de la classe émettrice de l'événement

Déclaration du type délégué qui précise la signature à respecter par les gestionnaires d'événements avec le mot clé **delegate**

Déclaration de l'événement avec le mot clé **event** du type du délégué.

Création de la méthode qui lève l'événement.

Ces méthodes peuvent être déclarées virtuelles pour faire l'objet d'une substitution dans une classe dérivée.

Si un gestionnaire d'événement a été assigné, il est alors exécuté.

Cela est exprimé par l'instruction `if event non nul alors execute event`.

Il faut alors qu'un souscripteur se soit abonné à l'événement.

```
public delegate void ChangementPrenomEventHandler(object sender, EventArgs e);
public event ChangementPrenomEventHandler ChangementPrenom;
1 référence
protected virtual void OnChangementPrenom(EventArgs e)
{
    if (ChangementPrenom != null) ChangementPrenom(this, e);
}
```

Depuis les dernières versions du langage vous trouverez aussi cette forme avec utilisation de l'opérateur conditionnel `?.` :

```
ChangementPrenom?.Invoke(this, e);
```

Invocation de la méthode qui lève l'événement sur changement de prénom.

```
public string Prenom
{
    get { return (this._prenom); }
    set
    {
        if (_prenom != string.Empty && _prenom != value)
            OnChangementPrenom(new EventArgs());
        this._prenom = value;
    }
}
```

4.2. Au niveau de la classe consommatrice de l'événement

Nous avons créé un objet de type personne.

Nous assignons un gestionnaire d'événement à l'événement changement de nom.

```
personne.ChangementPrenom += Personne_ChangementPrenom;
```

Le gestionnaire d'événement doit respecter la signature du délégué.

Cela peut aussi s'écrire ainsi :

```
personne.ChangementPrenom +=  
    new Personne.ChangementPrenomEventHandler(Personne_ChangementPrenom);
```

Le gestionnaire qui sera exécuté :

```
private void Personne_ChangementPrenom(object sender, EventArgs e)  
{  
    ...  
    MessageBox.Show("Prénom Modifié");  
}
```

4.3. Exemple du changement de nom d'une personne

Nous souhaitons transporter les informations ancien nom et nouveau nom à destination de l'application cliente qui traitera l'événement.

Nous devons créer un type (une classe) dérivée de la classe EventArgs.

Si ce ne sont les règles particulières de nommage, elle se construit comme tout autre classe.

```
public class ChangementNomEventArgs : EventArgs  
{  
    private string _ancienNom;  
    private string _nouveauNom;  
  
    public ChangementNomEventArgs(string ancienNom, string nouveauNom)  
    {  
        _ancienNom = ancienNom;  
        _nouveauNom = nouveauNom;  
    }  
  
    public string AncienNom  
    {  
        get { return _ancienNom; }  
        set { _ancienNom = value; }  
    }  
  
    public string NouveauNom  
    {  
        get { return _nouveauNom; }  
        set { _nouveauNom = value; }  
    }  
}
```

Au niveau de la classe **Personne** Emetteur de l'événement

Déclaration du type délégué. La signature du délégué comporte deux arguments qui représentent respectivement l'objet émetteur de l'événement **sender** et les données transportées avec l'événement **e**.

```
/// <summary>
/// Type Délégué d'événement
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
public delegate void ChangementNomEventHandler(object sender, ChangementNomEventArgs e);
```

Création de l'événement du type du délégué :

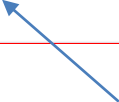
```
/// <summary>
/// Déclaration de l'événement de type délégué
/// </summary>
public event ChangementNomEventHandler ChangementNom;
```

Invocation de la méthode qui génère l'événement lors de la modification du champ :

```
public string Nom
{
    get { return (this._nom); }
    set
    {
        if (_nom != string.Empty && _nom != value)
            OnChangementNom(new ChangementNomEventArgs(this._nom, value));
        this._nom = value;
    }
}
```

Méthode de déclenchement de l'événement :
:

```
protected virtual void OnChangementNom(ChangementNomEventArgs e)
{
    ChangementNom?.Invoke(this, e);
}
```

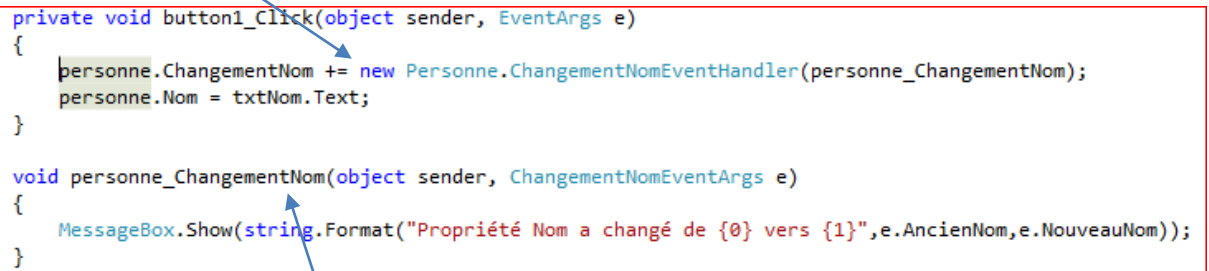


La(les) méthode(s) référencée(s) dans le délégué est alors invoquée.

Au niveau de la classe Consommateur de l'événement

Mise en place d'un gestionnaire d'événement :

Abonnement



```
private void button1_Click(object sender, EventArgs e)
{
    personne.ChangementNom += new Personne.ChangementNomEventHandler(personne_ChangementNom);
    personne.Nom = txtNom.Text;
}

void personne_ChangementNom(object sender, ChangementNomEventArgs e)
{
    MessageBox.Show(string.Format("Propriété Nom a changé de {0} vers {1}", e.AncienNom, e.NouveauNom));
}
```

Méthode exécutée lorsque l'événement survient

5. Déclaration d'un événement avec un délégué d'événement spécialisé

5.1. Au niveau de la classe Emetteur

L'architecture .Net propose un modèle d'événement s'appuyant sur un délégué spécialisé et les génériques.

Cette approche évite la déclaration spécifique d'un délégué.

Il est alors possible d'utiliser la forme suivante qui fait appel au délégué spécialisé **EventHandler** lorsque l'événement ne transporte pas de données en arguments.

```
public event EventHandler ChangementDateNaissance;
```

Si vous souhaitez créer un événement accompagné d'informations particulières, il vous faudra alors utiliser le modèle générique d'EventHandler, **EventHandler<T>** ou **T est une classe dérivée d'EventArgs**.

Dans l'exemple du changement de nom, nous souhaitons connaître l'ancien nom et le nouveau nom.

```
public event EventHandler<ChangementNomEventArgs> ChangementNom;
```

Les autres éléments restent inchangés.

5.2. Au niveau de la classe Récepteur Exe Console

Le mécanisme d'abonnement et d'association du gestionnaire d'événement qui sera invoqué.

Au niveau de la classe **Application** ici un programme en mode console.

```
Console.WriteLine("Création d'une personne puis déclenchement de l'événement");
Personne personne = new Personne() { Nom = "boste" };

personne.ChangementNom += Personne_ChangementNom;
// La modification de la propriété va déclencher l'événement
// et provoquer l'exécution de la méthode affectée à ChangementNom
personne.Nom = "Bost";
```

Le gestionnaire d'événement défini dans l'application :

```
static void Personne_ChangementNom(object sender, ChangementNomEventArgs e)
{
    Console.WriteLine("Propriété Nom a changé de {0} vers {1}", e.AncienNom, e.NouveauNom);
    Console.ReadLine();
}
```

```
Création d'une personne puis déclenchement de l'événement
Propriété Nom a changé de boste vers Bost
```