6

Délégués et traitement d'événements

Dans ce chapitre, nous allons nous intéresser aux événements. Ceux-ci jouent un rôle considérable en programmation Windows et Web : lorsqu'il se passe quelque chose dans un programme (par exemple un clic sur un bouton mais aussi pour bien d'autres motifs), Windows nous signale un événement. Et pour cela, il appelle une fonction bien particulière du programme.

Un programme, qu'il soit Web ou Windows, consiste essentiellement à traiter des événements.

Événements et traitements d'événements ne sont cependant pas propres ou liés exclusivement à la programmation Windows ou Web. Il s'agit, avec les délégués et les événements, de mécanismes généraux dont n'importe quel programme peut tirer profit.

Introduisons d'abord le sujet avec les délégués puisque les événements sont fondés sur eux.

C# version 2 a également introduit les fonctions anonymes (voir la section 6.3).

6.1 Les délégués

Un délégué (*delegate* en anglais) est un objet (en fait, de la classe `Delegate` ou d'une classe dérivée de celle-ci, mais peu importe) qui permet d'appeler une fonction, qui n'est pas nécessairement toujours la même. Il peut même s'agir d'un appel à une série de fonctions. Un délégué présente des similitudes avec les pointeurs de fonction du C/C++ mais

permet d'aller plus loin (car un délégué peut exécuter plusieurs fonctions les unes à la suite des autres) tout en faisant preuve de moins de laxisme.

Pour ceux qui connaissent le C et/ou C++, rappelons qu'un pointeur de fonction désigne une variable qui fait référence à une fonction. À partir d'un pointeur de fonction, on peut appeler la fonction pointée.

Prenons un exemple simple, que nous commenterons ensuite :

Les délégués Ex1.cs

```
using System;
class Prog
{
 static void f() {Console.WriteLine("Fonction f");}
 static void g() {Console.WriteLine("Fonction g");}
 delegate void T();       // ceci est une définition de type
 static void Main()
 {
  T de = new T(f);        // déclaration de variable. de fait référence à f
  de();                      // exécute f
  de = new T(g);          // de fait maintenant référence à g
  de();                   // exécute g
 }
}
```

Rien de nouveau en ce qui concerne les fonctions `f`, `g` et `Main`. Elles sont qualifiées de `static` car, pour simplifier, nous ne créons aucun objet.

Nous créons ensuite un nouveau type, appelé ici `T`. Ne vous fiez pas aux apparences : `T` désigne un type d'objet (au même titre que `int`, `double` ou une classe) et non une déclaration de variable, une signature de fonction, ou quoi que ce soit d'autre, comme un coup d'œil rapide pourrait le faire croire. Dans la définition de `T`, nous avons le choix de son nom (`T`) et nous signalons, avec le mot réservé `delegate`, qu'il s'agit d'un type délégué. `de` est donc une variable de type délégué. On parle plus communément de délégué pour `de`.

Un objet de type `T`, comme `de`, peut faire référence à une ou plusieurs fonctions, mais avec cette restriction : ces fonctions doivent respecter la signature reprise dans la définition qu'est `delegate void T()`. Ici : aucun argument (parenthèses vides) et aucune valeur de retour (`void`). C'est pour cette raison que la ligne `delegate` ressemble à une signature de fonction.

Il devient alors possible d'exécuter cette ou ces fonctions via le délégué.

Avec :

```
T de = new T(f);
```

nous déclarons et créons une variable baptisée `de` et de type `T`. Comme nous spécifions la fonction `f` en argument du constructeur, notre objet `de` fait référence à la fonction `f`.

Les programmeurs C et C++ diraient (et cela reste vrai en C#) que `de` pointe sur `f`. Pour faire exécuter cette fonction `f` via le délégué, il suffit d'écrire :

```
de();                  // exécution de la fonction pointée
```

Si nous écrivons maintenant :

```
de = new T(g); de();
```

`de` contient désormais une référence à la fonction `g` (et plus à `f`) et `de()` fait maintenant exécuter `g`.

Depuis la version 2, il est possible d'écrire (sans devoir créer explicitement une fonction, voir à ce sujet les fonctions anonymes à la section 6.3) :

```
T d = delegate {Console.WriteLine("Dans fonction anonyme");};
.....
d();
```

Un délégué peut faire référence à plusieurs fonctions, et ainsi les exécuter les unes à la suite des autres (on parle alors de *multicasting delegate*). À tout moment, il est en effet possible d'ajouter, ou de retirer, des fonctions à la liste des fonctions référencées par un délégué. Par exemple :

```
de = new délégué(f);       // de contient une référence à f
de += new délégué(g);      // de contient une référence à f et une autre à g
de();                      // exécution de f et puis de g
de -= new délégué(f);      // de ne contient plus qu'une référence à g
de();                      // exécution de g
de -= new délégué(g);      // de ne contient plus aucune référence de fonction
if (de != null) de();
else Console.WriteLine("Aucune fonction à exécuter");
```

Notre variable `de` prend finalement la valeur `null` quand plus aucune fonction ne lui est liée.

Le programme s'écrit de la même façon quand le délégué porte sur une fonction présentant une signature différente, avec arguments et valeur de retour :

Délégué pour fonctions acceptant un entier en argument et renvoyant un réel `Ex2.cs`

```
using System;
class Prog
{
 static double f(int n) {return n/2.0;}
 static double g(int n) {return n*2;}
 delegate double délégué(int n);    // délégué désigne un type d'objet
 static void Main()
 {
  délégué de;                       // déclaration de variable
  de = new délégué(f);              // de fait référence à f
  double d = de(5);                 // exécution de f, d prenant la valeur 2.5
  de = new délégué(g);              // de fait référence à g
  d = de(100);                      // exécution de g, d prenant la valeur 200
 }
}
```

ou si le délégué fait partie d'une classe distincte de celle du programme :

Délégué dans classe Ex3.cs

```
using System;

class A
{
 public void f() {Console.WriteLine("Fonction f de A");}
 public void g() {Console.WriteLine("Fonction g de A");}
 public delegate void del();             // définition de type délégué
}

class Prog
{
 static void Main()
 {
  A a = new A();
  A.del de = new A.del(a.f);
  de();                                  // exécution de f de A
  de = new A.del(a.g);
  de();                                  // exécution de g de A
 }
}
```

Avec :

```
public delegate void del();
```

on définit un type dans `A` (c'est comme si on y avait défini une classe). On n'a donc pas déclaré de champ membre de `A`. Avec :

```
A.del de = new A.del(a.f);
```

on crée une variable du type défini dans `A`, et celle-ci (qui est un délégué) fait référence à `f` de `A`.

Depuis la version 2, il est possible d'écrire (sans devoir créer explicitement une fonction, voir à ce sujet les fonctions anonymes à la section 6.3) :

```
T d = delegate(int n) {Console.WriteLine("Dans fonction anonyme avec " + n);};
.....
d(10);
```

6.2 Les événements

Nous savons que les événements jouent un rôle considérable dans les programmes Web ou Windows. Quand vous cliquez sur un bouton, du code (Windows dans les programmes Windows, mais ASP.NET pour les programmes Web) détecte le clic en

un emplacement de la fenêtre. Comme il se passe quelque chose, on parle d'événement (*event* en anglais).

Windows doit alors informer votre programme qu'un clic a été détecté. Pour cela, Windows appelle une fonction de votre programme. Une telle fonction s'appelle « fonction de traitement de l'événement clic » (*event handler,* ou plus simplement *handler* en anglais) et doit avoir une signature bien précise :

```
void xyz(object sender, EventArgs e);
```

De quelle fonction s'agit-il ? C'est vous qui le décidez en associant une fonction à l'événement, comme on le fait pour un délégué :

```
bTest.Click += new System.EventHandler(bTest_Click);
.....
private void bTest_Click(object sender, EventArgs e)
{
 .....
}
```

`bTest` désigne un objet de la classe `Button` qui comprend un objet d'un type particulier : `event`. De même que les délégués, `event` désigne un type particulier de classe, en rapport évidemment avec des événements. `EventHandler` désigne une classe de type `event`.

Depuis la version 2, la première ligne pourrait être écrite :

```
bTest.Click += bTest_Click;
```

On sait maintenant qu'un objet de type délégué peut faire référence à une ou plusieurs fonctions (la signature de ces fonctions devant correspondre à celle du délégué) et qu'il est possible d'exécuter ces fonctions via le délégué.

Le type `event` est fondé sur le type délégué. Un objet de type `event` (cas de `Click` ci-dessus) peut faire référence à une voire plusieurs fonctions de traitement d'événements, celles-ci devant avoir une signature bien particulière. Comme pour les délégués, ces fonctions peuvent être exécutées via la variable de type `event`.

Ici, on a associé la fonction `bTest_Click` à l'événement `Click`.

Une fonction de traitement d'événements doit avoir le format suivant :

- en premier argument, l'objet qui est à l'origine de l'événement ;
- en second argument, un objet `EventArgs` (ou d'une classe dérivée de celle-ci) qui contient généralement des informations complémentaires sur l'événement.

Le mécanisme des événements n'est pas limité à la programmation Windows et est bien plus général. Il est même indépendant de tout phénomène physique.

Pour mettre en place un mécanisme d'événements, il faut du code qui détecte un événement (de manière générale, quelque chose qui se passe, qui commence ou se termine). Ce code signale l'événement à d'autres objets. Encore faut-il que ceux-ci soient inscrits pour être informés de l'événement (on dit qu'ils sont notifiés).

Prenons un exemple pour illustrer les événements mais sans nous placer dans le cadre Windows. Nous aurons dans notre programme (par analogie avec un exemple tiré de la vie de tous les jours) :

- Un objet `surveillant` de la classe `Surveillant`. C'est lui qui détecte en premier l'événement (il est donc à la source du mécanisme). En cas d'accident, il avertit le Samu. Évident peut-être pour nous comme réaction mais pas pour un programme. Comment le surveillant peut-il savoir qu'il doit appeler le Samu en cas d'accident ? `samu` doit pour cela s'inscrire auprès de `surveillant` et demander à être notifié en cas d'accident. La classe `Surveillant`, de son côté, doit contenir un champ de type `event` pour permettre cet enregistrement.
- Un objet `samu` de la classe `Samu`. Nous venons de voir que celui-ci ne doit avertir `surveillant` qu'en cas d'accident (d'événement, de manière générale), et une fonction bien particulière de la classe `Samu` doit être appelée.
- On parle de fonction de traitement d'événements mais aussi de fonction de rappel (*callback* en anglais). Cet appel de la fonction de traitement est effectué à l'initiative de `surveillant` au moment où celui-ci détecte l'événement.
- À tout moment, `samu` pourrait avertir `surveillant` qu'il suspend, jusqu'à nouvel ordre, toute demande de notification (`samu` se retire pour cela de la liste de notification maintenue par `surveillant`). D'autres acteurs que `samu` (par exemple les journalistes) pourraient demander à `surveillant` d'être également placés sur la liste de notification. Pour simplifier, nous supposerons que seul `samu` doit faire l'objet d'une notification.

Quand survient un événement, `surveillant` doit certes appeler les fonctions de traitement des objets ayant réclamé une notification, mais il doit aussi fournir des informations quant à l'événement (nature et localisation de l'accident, par exemple). À cet effet, `surveillant` crée un objet de la classe `EventArgs` ou d'une classe dérivée de celle-ci (il pourra ainsi ajouter des champs qui sont propres à l'événement). Il faudra donc créer une classe qui est dérivée de `EventArgs` : soit `AccidentEventArgs` cette classe dérivée (il est d'usage de laisser le suffixe `EventArgs`). Pour simplifier, nous laissons le champ `Adresse` en champ public plutôt qu'en propriété :

```
class AccidentEventArgs : EventArgs
{
 public string Adresse;
 public AccidentEventArgs(string a) {Adresse = a;}
}
```

Intéressons-nous maintenant à la classe `Surveillant`. Celle-ci doit pouvoir appeler zéro, une ou plusieurs fonctions (selon le nombre de demandes de notification) en cas d'événement. La classe doit donc comporter une variable de type `event`, que nous décidons d'appeler `Accident`. Mais une variable de type `event` doit s'appuyer sur un délégué, mentionné d'ailleurs dans la déclaration de la variable événement. Il est d'usage, pour le nom du type délégué, de reprendre le nom de l'événement suffixé de `Handler` :

```
public delegate void AccidentHandler(object sender, AccidentEventArgs acc);
public event AccidentHandler Accident;
```

Comment appeler zéro, une ou plusieurs fonctions sans même les connaître au moment d'écrire le programme ? Grâce aux délégués ! Évidemment, ces fonctions de traitement d'événements ne doivent être appelées que si des fonctions ont été accrochées à la variable événement :

```
if (Accident != null) Accident(this, e);
```

L'objet `samu` doit s'enregistrer auprès de l'objet `surveillant` (sinon, `surveillant` ne sait pas qu'il doit avertir `samu`), en spécifiant la fonction à appeler :

```
class Samu
{
 // fonction de traitement d'événements
 public void onAccident(object sender, AccidentEventArgs e)
 {
  .....
 }
}
 // samu demande à être notifié
 surveillant.Accident += new Surveillant.AccidentHandler(samu.onAccident);
```

Présentons maintenant le programme dans son ensemble.

Traitement d'événements `Ex4.cs`

```
using System;
class AccidentEventArgs : EventArgs
{
 public string Adresse;
 public AccidentEventArgs(string a) {Adresse = a;}
}
class Surveillant
{
 public void Signaler(string adr)
 {
  AccidentEventArgs e = new AccidentEventArgs(adr);
  if (Accident != null) Accident(this, e);
 }
 public delegate void AccidentHandler(object sender, AccidentEventArgs acc);
 public event AccidentHandler Accident;
}
class Samu
{
 public void onAccident(object sender, AccidentEventArgs e)
 {
  Console.WriteLine("Appel reçu pour " + e.Adresse);
 }
}
```

Traitement d'événements *(suite)* — Ex4.cs

```
class Program
{
 static void Main(string[] args)
 {
  Surveillant surveillant = new Surveillant();
  Samu samu = new Samu();
  // samu demande à être notifié
  surveillant.Accident += new Surveillant.AccidentHandler(samu.onAccident);
  // simulation d'un accident
  surveillant.Signaler("Chernobyl");
 }
}
```

6.3 Les méthodes anonymes

C# version 2 a introduit les méthodes anonymes : au lieu de créer une fonction de traitement et de l'associer à un événement, comme cela devait se faire en versions 1 et 1.1, et comme cela peut encore se faire en version 2 (exemple ici du traitement du clic sur un bouton en programmation Windows ayant `bB` comme nom interne) :

```
// fonction de traitement (dans fichier Form1.cs)
private void bBonClick(object sender, EventArgs e)
{
 za.Text = "Il est " + DateTime.Now.ToLongTimeString();
}
.....
// associer la fonction à l'événement
// (opération effectuée dans le fichier Form1.Designer.cs)
bB.Click += new System.EventHandler(bBonClick);
```

Nous avons vu qu'il est maintenant possible d'écrire :

```
bB.Click += bBonClick;
```

Mais on peut aussi désormais écrire, en évitant de devoir créer explicitement une fonction de traitement :

```
bB.Click += delegate { za.Text = "Il est " + DateTime.Now.ToLongTimeString(); }
```

Il est possible de déclarer des variables à l'intérieur du `delegate`. Les variables sont alors locales au `delegate`. Il n'est cependant pas possible d'effectuer `-=` sur l'événement (pour mettre fin au traitement).

Programmes d'accompagnement

Les programmes `Ex1.cs` à `Ex4.cs` de ce chapitre.