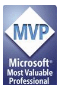
Association Dotnet France

Les nouveautés du langage C# 3.0

Version 1.4



James RAVAILLE
http://blogs.dotnet-france.com/jamesr

Sommaire

1 Introduction 3
1.1 Présentation 3
1.2 Pré-requis 3
2 Les accesseurs simplifiés 4
2.1 Présentation 4
2.2 Exemple de mise en œuvre 4
3 Les méthodes partielles 6
3.1 Présentation 6
3.2 Exemple de mise en œuvre 7
4 Les initialiseurs d'objets et de collections 9
4.1 Présentation 9
4.2 Exemple de mise en œuvre 9
4.2.1 Les initialiseurs d'objets 9
4.2.2 Les initialiseurs de collections 10
5 L'inférence de type et les types anonymes 11
5.1 Présentation de l'inférence de type 11
5.2 Présentation des types anonymes 11
5.3 Exemple de mise en œuvre 11
6 Les méthodes d'extension 13
6.1 Présentation 13
6.2 Exemple de mise en œuvre 14
6.3 Règles particulières 15
7 Les expressions lambda 17
7.1 Présentation 17
7.2 Exemple de mise en œuvre 17
7.3 Utilisation des expressions lambda comme paramètre de méthode 18
8 Conclusion 19

1 Introduction

1.1 Présentation

En Février 2008, Microsoft sort officiellement et pour le grand public, Visual Studio 2008, la version 3.5 du Framework .NET, ainsi qu'une nouvelle version du langage C#. Nous vous proposons dans ce cours, de vous présenter chacune des nouveautés de ce langage, avec pour chacune d'entre elles :

- Une partie théorique afin de vous expliquer en quoi elle consiste, et quel est son but, dans quels cas il est nécessaire/conseillé de l'utiliser…
- Une partie pratique avec des exemples de mise en œuvre.

Ces nouveautés sont les suivantes :

- Les accesseurs simplifiés.
- Les méthodes partielles.
- Les initialiseurs d'objets et de collections.
- L'inférence de type.
- Les types anonymes.
- Les méthodes d'extension.
- Les expressions lambda.

1.2 Pré-requis

Avant de lire ce cours, vous devez avoir lu les précédents cours sur le langage C# :

- Le langage C#.
- La programmation orientée objet avec le langage C#.
- Les nouveautés du langage C# 2.0.

2 Les accesseurs simplifiés

2.1 Présentation

Les accesseurs (ou propriétés) simplifiés, offrent la possibilité de définir, au sein d'une classe, des propriétés sans implémenter les attributs qu'ils « gèrent », ni le corps de l'accesseur en lecture (getter) et l'accesseur en écriture (setter). En effet, en rétro-compilant le code obtenu, on peut observer que l'attribut géré est généré automatiquement, tout comme le code des accesseurs.

Quelques « restrictions » toutefois, par rapport aux accesseurs écris avec le langage C# 2.0 :

- Les accesseurs en lecture et en écriture, doivent obligatoirement être tous les deux présents. Il n'est pas possible de définir un accesseur en lecture ou écriture seule.
- Les accesseurs en lecture et en écriture ont le même niveau de visibilité : impossible de réduire le niveau de visibilité de l'accesseur, en lecture ou en écriture.
- Le type de l'attribut généré est obligatoirement celui de l'accesseur.

L'intérêt des accesseurs simplifiés est d'écrire moins de code dans les classes.

2.2 Exemple de mise en œuvre

Soit la classe Voiture suivante :

```
// C#

public class Voiture
{
    #region Attributs et accesseurs

        private string _NumeroImmatriculation;
        public string NumeroImmatriculation
        {
            get { return this._NumeroImmatriculation; }
            set { this._NumeroImmatriculation = value; }
        }

        public string Marque
        {
            get;
            set;
        }

    #endregion
}
```

La classe ci-dessus présente :

- Un attribut avec son accesseur (aussi appelé accesseur « par défaut »), contenant le numéro d'immatriculation d'une voiture.
- Un autre accesseur avec une forme particulière, contenant la marque d'une voiture : les accesseurs en lecture et en écriture ne sont pas implémentés. Il s'agit d'un accesseur simplifié. Au final, lors de la génération du code MSIL, notre classe possède deux attributs, et pour chacun d'entre eux, un accesseur. Pour l'observer, il suffit de « rétro-compiler » le code

MSIL de cette classe, avec un outil tel que Reflector (http://www.red-gate.com/products/reflector). Voici ce qu'on peut observer :

```
public class Voiture
{
    // Fields
    private string _NumeroImmatriculation;
    [CompilerGenerated]private string <Marque>k__BackingField;

    // Properties
    public string Marque {
        [CompilerGenerated] get;
        [CompilerGenerated] set;
    }
    public string NumeroImmatriculation {
        get;
        set;
    }
}
```

3 Les méthodes partielles

3.1 Présentation

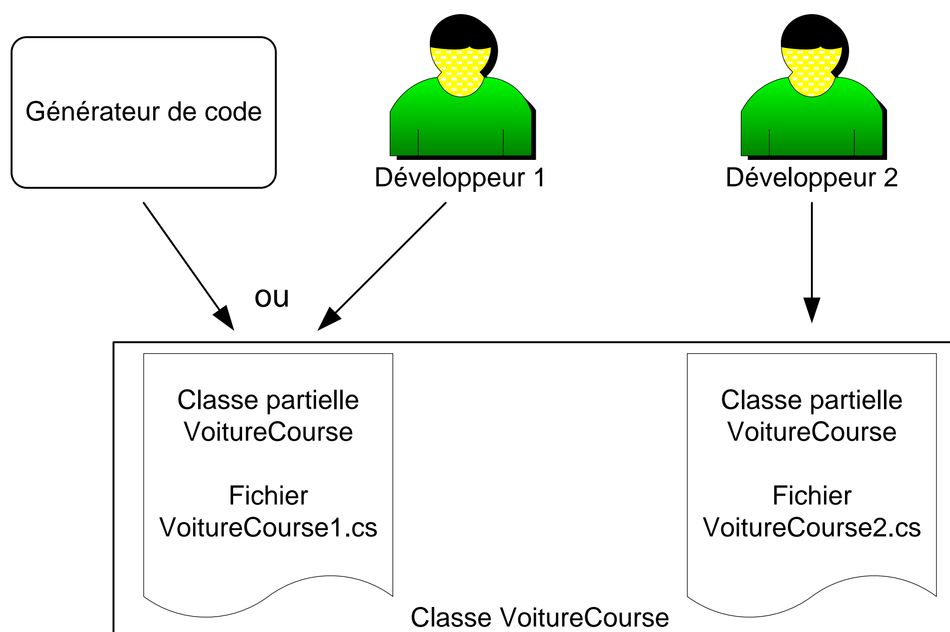
Les méthodes partielles sont obligatoirement contenues dans des classes partielles (nouveautés du langage C# 2.0), elles-mêmes contenues dans le même projet. Pour rappel, deux classes partielles constituent la même classe, séparées physiquement dans deux fichiers ou plus. Une méthode partielle est une méthode dont :

- La signature de la méthode est définie dans une classe partielle.
- L'implémentation de la méthode est définie dans une autre partie de la classe partielle. Cette implémentation est optionnelle.

Voici les caractéristiques des méthodes partielles :

- Elles sont obligatoirement privées. Elles ne peuvent être appelées que dans la classe dans laquelle elles sont définies. Aucun niveau de visibilité ne doit être spécifié (une erreur de compilation survient, le cas échéant).
- Elles sont définies avec le mot clé *partial*.

L'intérêt majeur des classes partielles réside dans les générateurs de code, ou bien encore dans des projets qui partagent des assemblies communes :



Par exemple, via un générateur de code (prenons par exemple LINQ for SQL), on génère une classe. Cette classe contient des membres pour lesquels il n'est pas possible de définir une implémentation lors de la génération. Le générateur de code ne génère alors que la signature de la méthode, et les appels de cette méthode dans la même classe. Cette implémentation ne peut être fournie que par le développeur 2. Ce dernier, au lieu de la fournir dans le fichier généré (ces

modifications pourraient être perdues lors d'une prochaine génération de la classe), va la fournir en définissant la même méthode partielle dans une autre partie de la classe partielle.

3.2 Exemple de mise en œuvre

Voici un exemple de méthodes partielles :

```
// C#

partial class VoitureCourse
{
    #region Attributs et accesseurs
        private string _NumeroImmatriculation;
        public string NumeroImmatriculation
        {
            get { return _NumeroImmatriculation; }
            set { _NumeroImmatriculation = value; }
        }

        public string Marque
        {
            get;
            set;
        }
    #endregion

    #region Constructeurs
        public VoitureCourse(string aNumeroImmatriculation, string 
aMarque)
        {
            this.NumeroImmatriculation = aNumeroImmatriculation;
            this.Marque = aMarque;

            this.Demarrer();
        }
    #endregion

    #region Methodes
        partial void Demarrer();
    #endregion
}
```

La méthode *Demarrer* est une méthode partielle. Cette méthode est appelée dans le constructeur de la même classe. Nous avons vu précédemment, que l'implémentation de la méthode partielle dans l'autre classe partielle est optionnelle. Si on observe le code C# rétro-compilé de l'assembly contenant cette classe avec l'outil Reflector (http://www.red-gate.com/products/reflector), alors on peut observer que l'appel de la méthode *Démarrer*, est présent uniquement si l'implémentation est réalisée dans une autre partie de la classe partielle.

Voici un exemple d'implémentation de la méthode *Démarrer*. Cette méthode aussi ne doit pas être définit avec un niveau de visibilité, et doit avoir la même signature que la méthode précédemment définie :

```
// C#

partial void Demarrer()
{
    Console.WriteLine("La voiture " + this.NumeroImmatriculation + " a 
démarré");
}
```

4 Les initialiseurs d'objets et de collections

4.1 Présentation

Le rôle des initialiseurs d'objets, est de « simplifier » l'écriture de la création d'objets à partir d'une classe ou d'un type anonyme, en combinant dans la même instruction :

- L'instruction de la création de l'objet.
- L'initialisation de l'état de l'objet (soit l'ensemble des attributs).

Les initialiseurs de collection permettent de définir quels sont les éléments qui constituent cette collection, lors de son instanciation. Ils permettent ainsi de simplifier la création et l'alimentation d'une collection de données ou d'objets.

4.2 Exemple de mise en œuvre

4.2.1 Les initialiseurs d'objets

Soit la classe *Voiture* suivante :

```
// C#

partial class VoitureCourse
{
    #region Attributs et accesseurs
        private string _NumeroImmatriculation;
        public string NumeroImmatriculation
        {
            get { return _NumeroImmatriculation; }
            set { _NumeroImmatriculation = value; }
        }

        public string Marque
        {
            get;
            set;
        }
    #endregion

    #region Constructeurs
        public VoitureCourse(string aNumeroImmatriculation)
        {
            this.NumeroImmatriculation = aNumeroImmatriculation;
            this.Marque = string.Empty;

            this.Demarrer();
        }

        public VoitureCourse() : this(string.Empty)
        {
        }
    #endregion

    #region Methodes
        partial void Demarrer();
    #endregion
}
```

Voici un bloc d'instruction C#, permettant de créer une instance de cette classe, en utilisant un constructeur défini dans la classe, et l'initialiseur d'objets :

```
// C#

Voiture oVoiture ;

oVoiture = new Voiture() {
      Marque = "Renault",
      NumeroImmatriculation = "212 YT 44"
};
```

Via l'initialisation d'objet, il est ainsi possible dans la même instruction, de créer une instance de la classe *VoitureCourse* en utilisant l'un des deux constructeurs présents dans la classe, et d'initialiser tous les attributs souhaités. A noter que le constructeur est toujours appelé avant l'initialisation des attributs.

4.2.2 Les initialiseurs de collections

Voici un exemple, montrant commun créer une collection de nombre entiers :

```
// C#

List<int> oListeEntiers = new List<int>() { 1, 3, 4, 6, 90, 34 };
```

Voici un autre exemple, montrant comment créer une collection d'objets de type *VoitureCourse* :

```
// C#

List<Voiture> oListeVoitures = new List<VoitureCourse>() {
    new VoitureCourse ("34 YT 54"),
    new VoitureCourse ("17 RE 33"),
    new VoitureCourse ("106 IU 75") {Marque="Peugeot"}
};
```

L'exemple ci-dessus, vous montre qu'il est aussi possible d'utiliser l'initialisation d'objets, dans une instruction utilisant l'initialisation de collections.

5 L'inférence de type et les types anonymes

5.1 Présentation de l'inférence de type

L'inférence de type permet de déclarer une variable locale sans préciser son type. Cependant, cette variable doit obligatoirement être initialisée, afin que le compilateur puisse déterminer son type, à partir de la valeur ou l'expression d'initialisation. Une fois la variable déclarée, on peut lui appliquer tous les membres, exposés par le type automatiquement déterminé.

Je pense que l'inférence de type, doit uniquement être utilisée quand il n'est pas possible ou très difficile de déterminé le type de la variable à partir de sa valeur d'initialisation. Cette situation peut se produire dans deux cas :

- Lorsqu'un type anonyme est utilisé dans la valeur d'initialisation. C'est le cas lors de la réalisation de projection de données dans les requêtes LINQ.
- Lorsque le type de la valeur d'initialisation est complexe, et donc difficilement définissable par tout développeur.

L'inférence de type est mise en œuvre au travers du mot clé *var*.

5.2 Présentation des types anonymes

Le langage C# 2.0 introduisait les méthodes anonymes. Le langage C# 3.0 introduit les types anonymes. Il n'y a pas de relation directe entre ces deux notions.

Les types anonymes permettent de créer des objets et des collections d'objets, sans avoir à définir explicitement dans l'application, la classe utilisée pour créer les objets. C'est le compilateur qui se charge de générer dynamiquement la classe, lors de la compilation du code. L'utilisation de cette nouveauté entraine l'utilisation implicite de l'initialisation d'objets et de collections, ainsi que de l'inférence de types. On retrouve l'utilisation de toutes ces nouveautés dans l'écriture de requêtes LINQ.

Les types anonymes correspondent uniquement à des structures de données en mémoire. Ils ne peuvent contenir des méthodes, applicables aux objets qu'ils permettent de créer. Comme toutes classes, les types anonymes dérivent de la classe *System.Object*. Les seules méthodes alors applicables à ces objets sont ceux hérités de cette même classe.

La mise en œuvre des types anonymes est réalisée au travers de la syntaxe suivante : *new { propriété = valeur , … }*

5.3 Exemple de mise en œuvre

Voici un exemple de code, permettant de créer un objet nommé *oVoiture* à partir d'un type anonyme. La « structure » de cet objet est déterminée par les attributs qui sont définis et valorisés entre accolades. Cet exemple met en évidence un exemple d'utilisation des initialiseurs d'objets :

```
// C#

var oVoiture = new { Marque = "Renault",
                     NumeroImmatriculation = "212 YT 44",
                     Couleur = Color.Black };
```

Il est alors possible de créer des objets, contenant uniquement des données, sans avoir à implémenter dans le code la classe permettant de les créer. On remarque l'utilisation du mot clé *var*, marquant l'utilisation de l'inférence de type.

Voici un autre exemple, qui montre l'utilisation de l'inférence de type, d'un type anonyme et l'initialisation d'objets. La requête LINQ permet d'obtenir une collection d'objets, où chacun contient une marque de voiture et un numéro de département, à partir d'une collection de voitures. Seules les voitures immatriculées dans les départements 44 et 35, doivent être sélectionnées :

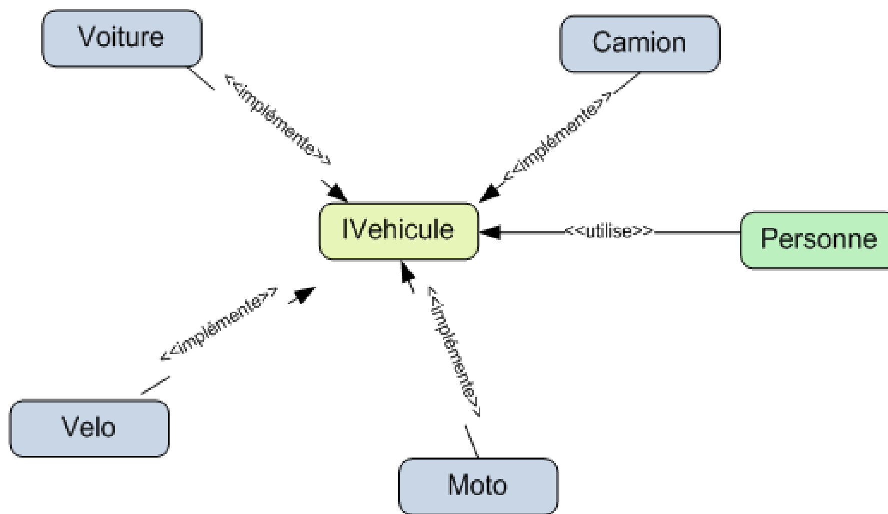
```
// C#

var oListeVoitures1 = (from oVoiture in oListeVoitures
                       where
oVoiture.NumeroImmatriculation.EndsWith("44") ||
oVoiture.NumeroImmatriculation.EndsWith("35")
                       select new {
                           Marque = oVoiture.Marque,
                           Departement =
oVoiture.NumeroImmatriculation.Substring(oVoiture.NumeroImmatriculation.L
ength - 2)
                       }).ToList();
```

6 Les méthodes d'extension

6.1 Présentation

Les méthodes d'extension permettent d'étendre une classe, en ajoutant de nouvelles méthodes, sans créer de classe dérivée de cette même classe. Elles sont utiles dans les cas suivants :

- Soit une interface *IVehicule*, qui définit les membres de toutes classes permettant de créer des objets, qui se considèrent comme étant des véhicules. Soit les classes *Voiture*, *Camion*, *Velo*, *Moto*, qui implémentent cette interface.



Alors si on étend l'interface *IVehicule* par une méthode d'extension, alors cette méthode d'extension est applicable à toutes les instances, créées à partir des classes implémentant cette interface.

- Vous développez une application, qui utilise des classes contenues dans un assembly qui vous a été fourni. Vous ne possédez pas les sources de cet assembly. Vous savez qu'il contient une classe *Animal*, qui possède des classes dérivées. Vous souhaitez alors ajouter des membres supplémentaires à cette classe, afin d'enrichir les classes dérivées. Comme vous ne possédez pas les sources de l'assembly, vous ne pouvez le faire autrement qu'en créant des méthodes d'extension, qui étendent la classe *Animal*.

En C#, une méthode d'extension est une méthode statique, contenue dans une classe statique. Cette méthode sera utilisée :

- Comme toute méthode statique, en passant en paramètre l'objet à laquelle elle s'applique. Ainsi dans l'écriture du code, il est possible d'exécuter l'instruction suivante sans la levée d'une exception de type *NullReferenceException*, même si l'objet *oVoiture* est *null* ; la méthode *Garer* sera appelée, avec la valeur *null* en paramètre :

```
// C#

oVoiture.Garer();
```

- Comme toute autre méthode d'instance de la classe étendue.

La signature de cette méthode est particulière : le type du premier paramètre est préfixé du mot clé *this*, et ce type représente le type étendu. Une méthode d'extension peut posséder des paramètres d'entrée, qui sont alors à ajoutés à la suite de ce premier paramètre.

6.2 Exemple de mise en œuvre

Voici une méthode d'extension permettant d'étendre le type *System.Int32* du Framework .NET :

```
// C#

public static class IntExtension
{
    public static string MultiplierParDeux(this int aNombre)
    {
        return aNombre.ToString() + " * 2 = " + (aNombre * 2).ToString();
    }
}
```

Elle permet de multiplier par deux le contenu de toute variable de type *System.Int32*. Voici un exemple :

```
// C#

int i = 10;
Console.WriteLine(i.MultiplierParDeux());       => Affiche 10 * 2 = 20
```

Un autre exemple : toutes les classes représentant un dictionnaire de données, dit « générique » implémente l'interface *IDictionary<TKey, TValue>* du Framework .NET. Alors voici une méthode permettant d'étendre toute collection de données, implémentant cette interface, de manière à pouvoir accéder à la valeur d'un élément à partir de sa clé, ou obtenir une valeur par défaut, si aucun élément de la collection n'est identifié avec la clé :

```
// C#

public static class IDictionnaryExtension
{
    public static TValue GetValue<TKey, TValue>(
            this IDictionary<TKey, TValue> aListe,
            TKey key, TValue defaultValue)
    {
        TValue aValeurRetour;
        if (aListe.ContainsKey(key))
            aValeurRetour = aListe[key];
        else
            aValeurRetour = defaultValue;

        return aValeurRetour;
    }
}
```

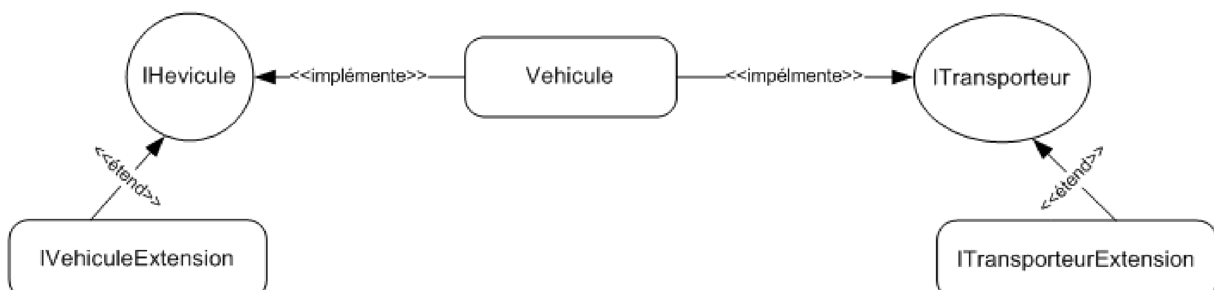
Et voici un exemple d'utilisation de la méthode d'extension, avec la classe générique *Dictionary<TKey, TValue>*, qui implémente l'interface générique *IDictionary<TKey, TValue>* :

```
// C#

// Création de la collection de données.
Dictionary<int, string> oListeFormations = new Dictionary<int, string>();
oListeFormations.Add(1, "Formation C#");
oListeFormations.Add(2, "Formation ASP .NET");
oListeFormations.Add(3, "Atelier Accès aux données");
oListeFormations.Add(4, "La sécurité dans les applications .NET");

// Utilisation de la méthode d'extension.
string sFormation = oListeFormations.GetValue(5, "formation
inexistante");
```

Le corps des méthodes d'extension peut être « paramétrables », « personnalisables », au travers de l'utilisation d'expressions lambda.

6.3 Règles particulières

Soit le diagramme de classes suivant :



Voici quelques règles à observer lors de l'utilisation de méthodes d'extension :

- Si les classes *IVehiculeExtension* et *ITransporteurExtension* possèdent une méthode d'extension de même nom et avec une signature « analogue », alors une erreur survient lors de la compilation, lorsque cette méthode est appliquée à une instance de la classe *Vehicule*.
- Si la classe *Vehicule* contient une méthode, et que la classe *IVehiculeExtension* et/ou *ITransporteurExtension* propose une méthode d'extension de même nom et avec une signature « analogue », alors l'application compile et la méthode de la classe *Vehicule* « masque » les méthodes d'extension.

7 Les expressions lambda

7.1 Présentation

Une expression lambda est une fonction ne possédant pas de nom (cela rappel les méthodes anonymes introduites dans le langage C# 2.0), exécutant un traitement et retournant une valeur. Une expression lambda est composée de trois parties :

- Une liste de paramètres
- L'opérateur =>
- Une expression

Dans quels cas peut-on utiliser les expressions lambda :

- Pour réaliser des fonctions de calcul.
- Largement utilisées dans les méthodes d'extension de l'interface *IEnumerable<T>*.

7.2 Exemple de mise en œuvre

Par exemple, vous manipulez une liste d'entiers. Et dans cette liste, vous souhaitez uniquement sélectionner les nombres pairs. Avec le langage C# 2.0, on écrira le bloc de code suivant :

```
// C#

List<int> oListeNombres = new List<int>() { 1, 34, 3, 9, 12, 18};
List<int> oListeNombrePaires = new List<int>();

foreach (int i in oListeNombres)
{
    if (i % 2 == 0)
        oListeNombrePaires.Add(i);
}
```

Avec le langage C# 3.0, en utilisant la méthode d'extension `Where` de l'interface générique *IEnumerable<T>*, on écrirait le bloc de code suivant :

```
// C#

List<int> oListeNombres = new List<int>() { 1, 34, 3, 9, 12, 18};
List<int> oListeNombrePaires;

oListeNombrePaires = oListeNombres.Where(i => i % 2 == 0).ToList();
```

Le bloc de code C# ci-dessus utilise la méthode d'extension *Where*, étendant l'interface générique *IEnumarable<T>* du Framework .NET. Cette méthode permet de filtrer la collection d'entiers, en fonction d'une expression booléenne. Cette expression booléenne est définie via l'expression lambda *i => i % 2 == 0*, qui signifie « pour tous les nombres i de la collection pour lesquels le résultat du modulo par deux vaut 0 ».

7.3 Utilisation des expressions lambda comme paramètre de méthode

Dans une méthode « classique » ou une méthode d'extension, il est possible de « paramétrer », « personnaliser » le comportement de la méthode, en utilisant une expression lambda.

Voici une méthode d'extension, permettant de parcourir une liste d'entiers à laquelle elle est appliquée, afin de les traiter. Le traitement de ces entiers n'est pas déterminé dans la méthode d'extension elle-même, mais par le code utilisant cette méthode d'extension :

```
// C#

public static void TraiterElements(this List<int> aListe, Func<int, int>
aExpression)
{
    for (int i=0; i<aListe.Count; i++)
    {
        aListe[i] = aExpression(i);
    }
}
```

Pour définir une expression lambda, il suffit d'utiliser le mot clé *Func* permettant de définir des types de données (une évolution des délégués). Dans l'exemple précédent, *Func<int, int>* décrit une expression de traitement acceptant une donnée de type *int* en paramètre (le premier type précisé), effectuant un traitement avec ou sur ce paramètre, et retournant une donnée de type *int* (second type de donnée défini).

Enfin, voici deux blocs de code utilisant la méthode *TraiterElements* sur une collection d'entiers, afin de multiplier par 2 chacun des nombres de la collection :

```
// C#

// Premier exemple.
Func<int, int> oExpressionTraitement = (i) => i * 2;
oListeNombres.TraiterElements(oExpressionTraitement);

// Second exemple.
oListeNombres.TraiterElements((i) => i * 2);
```

8 Conclusion

Ce chapitre vous a présenté les nouveautés du langage C# 3.0. Ces nouveautés ont été nécessaires, afin de permettre d'écrire des requêtes LINQ (sur une grappe d'objets, sur une base de données SQL Server, ou sur un flux XML), et l'utilisation de composants d'accès aux données tels que Classes LINQ For SQL.

Voici un exemple, qui récapitule et combine de nombreuses nouveautés du langage C# 3.0, au travers d'une requête LINQ : écrire une requête LINQ, permettant de sélectionner la liste des fichiers d'extension *txt* d'un répertoire, dont la taille est strictement supérieure à 10 Ko. En sortie, on doit obtenir uniquement le nom et la taille des fichiers en octets, dans l'ordre alphabétique inversé sur le nom :

```
// C#

DirectoryInfo oDirInfo = new DirectoryInfo(@"c:\transfert");

// Sélection de la liste des fichiers (requête LINQ).
var oListeFichiers = oDirInfo.GetFiles("*.txt")
    .Where(oFichier => oFichier.Length > 10 * 1024)
    .OrderByDescending(oFichier => oFichier.Name)
    .Select(oFichier => new { oFichier.Name, Taille = oFichier.Length });

// Affichage de la liste des fichiers.
foreach (var oFichier in oListeFichiers)
{
    Console.WriteLine(oFichier.Name + " : " +
oFichier.Taille.ToString());
}
```

La requête LINQ présente dans le bloc de code ci-dessus, utilise pleinement les nouveautés du langage C# 3.0 que nous avons étudiées dans ce support :

- **Inférence de type** : liée à l'utilisation du type anonyme, et marquée par l'utilisation du mot clé *var*.
- **Méthodes d'extension** : marquées par l'utilisation des méthodes d'extension *Where*, *OrderByDescending*, et *Select* de l'interface générique *IEnumerable<T>* du Framework .NET.
- **Type anonyme** : marquée par l'utilisation du mot clé *new*, et la projection de données effectuée en ne sélectionnant que le nom et la taille du fichier en octets.
- **Expressions lambda** : utilisées dans les trois méthodes d'extension. Autrement dit, pour filtrer, trier les fichiers, et réaliser une projection de données sur les informations des fichiers.
- **Initialisation d'objet** : utilisée au sein du type anonyme.