

Les nouveautés du framework 2.0

par MORAND Louis-Guillaume ([Page perso de Louis-Guillaume MORAND](#))

Date de publication : 5/09/2005

Dernière mise à jour :

Présentation des nouveautés du framework dotnet 2.0

Introduction.....	3
1 - Les nouveautés du C#.....	4
1.1 - Les méthodes anonymes.....	4
1.2 - Les generics (ou classes templates).....	4
1.3 - Les itérateurs.....	5
1.4 - Les classes partielles.....	6
1.5 - Les types nullable.....	6
1.6 - L'opérateur ::.....	7
2 - Les nouveautés du VB.Net.....	8
2.1 - Les classes partielles.....	8
2.2 - Les types unsigned.....	8
2.3 - La surcharge d'opérateur.....	8
2.4 - Les instances par défaut.....	8
2.5 - Les blocks using.....	9
2.6 - L'opérateur IsNot.....	9
2.7 - L'objet My.....	9
2.8 - Le mot clé continue.....	10
2.9 - Les generics.....	10
3 - Les nouveautés de l'ASP.Net.....	11
4 - Nouveautés générales.....	12
Conclusion.....	13
Liens complémentaires.....	14
Remerciements.....	15
Téléchargements.....	16

Introduction

Alors que Visual Studio .Net montre le bout de son nez avec sa bêta 2, il est également possible de déjà découvrir les nouveautés du nouveau framework. Ce dernier ne sera pas une mise à jour mineure du framework 1.1 (et deviendrait donc le FX 1.2) mais bien une évolution conséquente; en effet, elle comprend des changements majeurs que ce soit dans ses concepts ou même simplement dans les langages de programmation.

Nous nous intéresserons ici, principalement aux nouveautés apportées aux langages de programmation (C#, VB.Net et ASP.Net) par cette nouvelle mouture du framework .Net.

1 - Les nouveautés du C#

1.1 - Les méthodes anonymes

Première nouveauté intéressante, les méthodes anonymes. Elles permettent de construire une méthode sans la nommer ou, plus clairement, passer un bloc de code en tant que paramètre.

Voyons deux cas simples:

```
// Méthode anonyme avec une seule instruction
monBouton.Click += new EventHandler(sender, e) {MessageBox.Show("test");};

// Méthode anonyme avec plusieurs instructions
monBouton.Click+= new EventHandler(sender, e)
{
    string test = "Le bouton appuyé est " + sender;
    Console.WriteLine(test + " avec l'event " + e);
}



// avant vous auriez eu
monBouton.Click+= new EventHandler(this.maMethode)
...
private void maMethode(object sender, EventArgs e)
{
    string test = "Le bouton appuyé est " + sender;
    Console.WriteLine(test + " avec l'event " + e);
}
```

Ainsi, pour des cas bien particuliers (par exemple lorsque vous savez que la méthode ne sera JAMAIS appelée autre part dans le code), il est possible d'appeler plusieurs lignes de code sans les mettre dans une méthode tierce. Ceci était déjà faisable en JAVA par exemple.

1.2 - Les generics (ou classes templates)

La première chose à dire, est que ces derniers sont une implémentation du concept de templates qui existait déjà en C++. Il est dorénavant supporté par C#, C++ et VB.Net.

Le principe même des Generics est la généralisation (et/ou réutilisation) de méthode sans ce soucier du typage des paramètres par exemples (transtypage).

Attention, les Generics ne sont pas identiques aux templates du C++ ( **Template** et  **templates C++**), ils ont encore un certain nombre de limitations:

- il n'est pas possible d'utiliser des opérateur arithmétiques mais on peut utiliser des opérateurs personnels
- les paramètres génériques ne peuvent avoir de valeur par défaut
- d'autres limitations dont l'utilisation très rare ne mérite pas de se pencher dessus pour le moment

Cela vous paraîtra sûrement plus clair à l'aide d'un exemple concret:

```
static void Main(string[] args)
{
    int a = 1, b = 5, c = 3;
    List<int> myIntegerList = new List<int>();
    myIntegerList.Add(a);
    myIntegerList.Add(b);
    myIntegerList.Add(c);
    foreach(int val in list)
    {
        total = total + val;
    }
}
```

Sans entrer dans les détails, nous verrons comment utiliser ce que l'on appelle une "Type-safe generic List", une **List** (collection) dont le type est sûr. En effet, une List peut, de base accepter des objets de type différents qui sont alors "boxés" (rangés sans être castés dans un type différent), ainsi lors de la récupération des "items" de la List, nous ne savons pas le type de ces derniers et le *unboxing* peut entraîner des erreurs d'exécution. Ici, nous "créons" une collection ne contenant que des objets du type Integer, et l'opération de boxing/unboxing n'a plus lieu d'être. Les opérations sur cette collection sont alors plus "sécurisées".

Bien sûr, il est possible de tirer avantage des *Generics* pour créer des classes réutilisables. En voici un exemple:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Developper
{
    public class ClasseDemo< Type1, Type2 >
    {
        private Type1 _obj;
        public ClasseDemo(KeyType obj1)
        {
            // constructeur
            _obj = obj1;
        }

        public Type2 Method(Type2 obj1, Type2 obj2)
        {
            // Code fonctionnel
        }
    }
}
```

Ce bout de code est un peu spécial, mais je tenais à montrer la possibilité d'utiliser plusieurs paramètres génériques pour une seule et même classe. Cette dernière prend donc deux paramètres, l'un étant utilisé dans son constructeur, l'autre compare deux objets et retourne par exemple l'un des deux selon une certaine comparaison.

Ainsi, vous savez maintenant que les *Generics* et leurs nombreuses utilisations vous permettront de réduire la taille de votre code et de faire des classes, méthodes réutilisables à volonté.

1.3 - Les itérateurs

Un itérateur est une méthode qui permet d'utiliser un "**foreach in**" sur une classe. Jusqu'à maintenant, vous utilisiez le `foreach in` sur des collections. Exemple:

```
foreach(ListViewItem lvi in maListView.Items)
{
}
```

Vous ne pouviez donc travailler que sur des objets de type Collection. C'est toujours le cas maintenant mais l'appel à ces collections se fait par l'appel à la classe parente de celles-ci.

Prenons comme exemple, la classe *Personne* qui permet de contenir le nom et les différents prénoms d'une personne.

```
public class Personne
{
    public string _Nom = "Morand";
    string[] _Prenoms = { "Louis-Guillaume", "Charles", "Aurélien" };
    public System.Collections.IEnumerator GetEnumerator()
    {
        foreach (string prenom in _Prenoms)
            yield return prenom;
    }
}

// utilisation de l'itérateur
```

```
// instantiation de la classe Personne
Personne p = new Personne();

// itération à travers le foreach
foreach (string prenom in p)
{
    Console.WriteLine(prenom);
}
```

Le mot-clé **yield** sert à retourner implicitement le type de l'objet retourné.

D'après la MSDN, les itérateurs ont les propriétés suivantes:

- une itérateur est une section de code retournant une suite ordonnée de valeurs
- un itérateur utilise le statement **yield return** pour renvoyer une valeur
- un itérateur peut être utilisé dans le corps d'une méthode ou dans un accesseur
- le type retourné par un itérateur doit être System.Collections.IEnumerable, System.Collections.IEnumerator ou l'un des itérateurs génériques

1.4 - Les classes partielles

Petite merveille, les classes partielles permettent de diviser une même classe dans plusieurs fichiers. Les "maniaques" du "une classe, un fichier" vont devoir revoir leur credo :).

Ces dernières sont (seront?) principalement utilisées dans les cas de travail en équipe. Sans parler de CVS ou autre, chacun pourra coder son propre fichier de classe partielle qui, au moment de la compilation, se "fusionnera" avec les autres fichiers de cette même classe partielle.

Prenons un exemple simple:

Classe normal passée en partielle (fichier du developpeur Louis-Guillaume)

```
partial class ClasseDeTest
{
    // une variable
    private string maVariable;

    public ClasseDeTest()
    {
        // mon constructeur
    }
}
```

Deuxième classe partielle (fichier du developpeur Laurent)

```
partial class ClasseDeTest
{
    //une methode
    public void maMethode()
    {
        MessageBox.Show(maVariable);
    }
}
```

Comme vous pouvez le voir, la méthode peut appeler des variables qui se trouvent dans la même classe mais dans un autre fichier. Dans un groupe de développeurs cela permet, entre autre, que chacun puisse créer séparément une partie d'une classe tout en pouvant utiliser les méthodes et variables codés par d'autres membres de l'équipe. Ou encore, séparer une grosse classe en la morcelant dans différents fichiers les membres, les constructeurs, les méthodes, etc.

1.5 - Les types nullable

Une variable de type *nullable* peut contenir toutes les valeurs possibles correspondant à son type, ainsi qu'une valeur additionnelle *null*. Ce problème résout entre autres le problème que vous pouviez rencontrer en travaillant avec des bases de données qui contenaient des variables *null* que vous ne pouviez "caster" correctement.

Prenons l'exemple le plus simple d'un entier (*Integer*) qui ne pouvait avec le Fx 1.1, être *null*.

```
int? monEntier = null;
```

Vous remarquez donc le caractère "?" à droite du type de l'objet que vous voulez *nullable*.

L'objet *nullable* se voit étendu de deux propriétés: *HasValue* qui retourne un booléen et *Value* qui retourne la valeur de l'objet. Ainsi, vous pouvez dorénavant tester la valeur d'un entier proprement:

```
if (monEntier.HasValue)
    MessageBox.Show("Valeur: " + monEntier.Value);
else MessageBox.Show("Valeur: nulle");
```

1.6 - L'opérateur ::

L'opérateur :: (ou qualificateur de namespace) permet d'appeler l'espace de nom global, lorsque ce dernier pourrait être caché par une entité (variable, méthode, propriété) locale du même nom.

Ce n'est pas clair donc voici un exemple montrant un cas particulier où son utilisation est nécessaire :

```
class MaClasse
{
    // définition d'une console nommée 'console'
    const int Console = 7;

    static void Main()
    {

        Console.WriteLine("Hello World"); // erreur: cela appelle MaClasse.Console


        // correction
        ::Console.WriteLine("Hello World");
    }
}
```

Bien entendu, libre au développeur de ne pas utiliser des mots clés réservés, mais dans certains cas, cela peut-être nécessaire.

2 - Les nouveautés du VB.Net

2.1 - Les classes partielles

Comme pour C#, il est maintenant possible d'utiliser les classes partielles en VB.Net 2.0. Pour des informations détaillées, reportez-vous au chapitre 1.4 (classes partielles en C#).

 *VB.Net, contrairement au C#, ne nécessite pas que toutes les "même" classes partielles aient le mot partial. Une (seule!!) d'entre elles peut ne pas l'avoir.*

2.2 - Les types unsigned

Avec le nouveau CLR, VB.Net peut maintenant utiliser les types entiers non-signés. Ces nouveaux types sont donc **UShort**, **UInteger**, **ULong** et **SByte**.

Pour ceux qui ne verraient pas l'intérêt de ses types, il faut savoir que l'utilisation de type non-signés pour stocker des entiers que l'on a par exemple, forcément positifs, permet d'avoir les meilleurs performances possibles sur des plateformes 32bits et donc pour votre application.


2.3 - La surcharge d'opérateur

La surcharge d'opérateur permet de donner aux opérateurs un comportement spécifique quand ils sont appliqués à des types spécifiques. Plus simplement, on peut définir le "comportement" qu'engendre l'utilisation d'un opérateur binaire sur deux objets d'un type défini.

Prenons un exemple on ne peut plus simple, la surcharge de l'opérateur "-" (soustraire).

```
Public Shared Operator -(ByVal str1 As MyString, _
    ByVal str2 As MyString) As String
    Return str1.Text.Substring _
        (0, str1.Text.Length - _
            str2.Text.Length) & str2.Text
End Operator
```

Les opérateurs surchargés remplacent avantageusement les appels de fonction, car ils nous évitent ici décrire une méthode effectuant la "soustraction" et retravaillant les deux chaînes de caractères

 *Les différents opérateurs qu'il est possible de surcharger sont: "+", "-", "*", "/", "^", "<", "<=", ">", ">=", "<<", ">>", "&", "Like", "Mod", "And", "Or", "Xor", "Not", "=", et quelques autres.*

2.4 - Les instances par défaut

Les programmeurs venant de VB6 se plaignaient souvent de devoir instancier une *Form* (en VB.Net) avant de l'afficher:

```
Dim maForm As New Form1
maForm.Show()
```

Ils peuvent dorénavant afficher une *Form* en utilisant son instance par défaut:

```
Form1.Show()
```

2.5 - Les blocks using

Encore une nouveauté copié sur le merveilleux C# (:D), le block **Using...End Using** permet de s'assurer de la libération des ressources de variables définies.

```
Public Sub maMethode()  
  
    Using conn As New SqlConnection(str)  
        Dim vari As String  
        vari= "developpez"  
        MsgBox(vari)  
    End Using  
  
    '-- vari est "nettoyé"  
End Sub
```

Le block **Using** assure, que dès que le programme "sort" du bloc **Using**, les ressources utilisées par l'objet vari, sont libérées.

2.6 - L'opérateur IsNot

Le nouvel opérateur **IsNot** est un mélange ingénieux des opérateurs Is et Not. Alors que vous deviez jusqu'à maintenant utiliser la nomenclature suivante:

```
If Not(monObjet Is Nothing) Then  
    MsgBox("monObjet n'est pas nul")  
End If
```

Vous utiliserez dorénavant:

```
If monObjet IsNot Nothing Then  
    MsgBox("monObjet n'est pas nul")  
End If
```

2.7 - L'objet My

L'objet **My**, petite exclusivité du VB.Net qui, à l'époque a donné l'expression "Il n'y a que My qui m'aïlle" (petite pique des développeurs VB.Net envoyés aux développeurs C#), permet au développeur d'accéder très rapidement à un petit nombre de nouveaux objets.

Nous avons:

- [My.Application](#) : donne accès au contexte de l'application, à sa culture, ses arguments, ses logs, le mode d'authentification, ou encore le splash screen de l'application
- [My.Computer](#): permet de jouer des fichiers wav, travailler sur le presse-papier, récupérer des informations sur le système (mémoire, etc) ou encore travailler sur le registre, le clavier, la souris les ports ou l'écran.
- [My.Forms](#): collection de tous les formulaires du projet
- [My.Ressources](#): permet de travailler sur différents types de ressources (audio, icônes, bitmaps,
- [My.Settings](#): permet de travailler sur les paramètres de l'application
- [My.User](#): permet d'obtenir des informations sur l'utilisateur (nom, groupe, domaine) mais également d'avoir des informations sur l'utilisateur principal de l'application mais aussi de définir cet utilisateur principal de l'application.
- [My.Webservices](#): fournit une instance de chaque Webservice du projet

Ronald Vasseur a écrit un article très complet sur le sujet et je vous encourage vivement à le lire: **VB.Net 2005 : le namespace My**

2.8 - Le mot clé continue

Mot clé largement utilisé dans différents langages comme le C++, ou le C#, le *continue* permet de passer à l'itération suivante d'une boucle sans exécuter le code "restant" dans cette boucle, c'est une rupture de séquence. Prenons un exemple simple mais concret:

```
For i As Integer = 0 To 10
    If i = 3 Then Continue For 'Si i=3, alors pas de message dans la console
    Console.WriteLine(i.ToString)
Next
```

Ce code présente donc le "Continue For", comme il existe le "Continue While" et le "Continue Do".

2.9 - Les generics

Il est simplement important d'indiquer que les Generics font aussi leur apparition en VB.Net.

3 - Les nouveautés de l'ASP.Net

Asp.Net 2.0 a lui aussi été doté d'un grand nombre de nouvelle fonctionnalités mais je vous laisse aller lire l'excellent article de **Didier Danse** qui présente plus en détails ces nouveautés: **Nouveautés d'Asp.Net 2.0**.

4 - Nouveautés générales

En sus des nouveautés énoncées précédemment et spécifique au C# et VB .NET, d'autres nouveautés concernant tout les langages portés sous .Net ont également vu le jour. Ces nouveautés sont à la fois technologiques et fonctionnelles. Nous allons en énumérer quelques unes.

Support du 64-bits

Dorénavant, le framework est capable de compiler des applications tirant profit des plateformes 64-bits, les rendant plus rapide et leur permettant d'utiliser plus de mémoire que les plateformes 32-bits

Utilisation du protocole FTP

L'utilisation du protocole et l'accès aux ressources FTP est maintenant facilité grâce à l'utilisation des classes WebRequest, WebResponse, et WebClient.

Amélioration de la console

L'utilisation de la console est améliorée. Il est maintenant possible de la contrôler plus facilement et ce, à l'aide de nombreuses nouvelles options (taille de la console, du buffer, tout simplement attendre une entrée utilisateur sur la console, contrôler les couleurs de la console, ducurseur, et bien d'autres choses).

Contrôle du réseau

Pour des applications travaillant en collaboration avec le réseau, il est maintenant possible de détecter des changements sur ce dernier grâce à la classe **NetworkChange**. Vous pourrez ainsi savoir lorsque vous êtes déconnecté, sorti d'une zone wireless ou encore un problème hardware.

Egalement l'arrivée de la classe Ping qui permet simplement de pinger une entité du réseau.

Enfin, grâce au namespace System.Net.NetworkInformation, il sera très facile d'obtenir des informations sur la configuration locale (ip, dns, etc)

Les Generics et collection génériques

Comme vu précédemment il est possible de créer des classes génériques mais il est également possible d'utiliser de nouvelles classes comme les collections génériques System.Collections.Generic.

IPv6 en remoting

Pour les développeurs utilisant le remoting, il est maintenant possible pour eux d'utiliser des adresses IPv6. De plus certaines classes de connexion permettent d'utiliser l'encryptage et l'authentification SSPI(Security Support Provider Interface). Enfin, il est possible grâce au namespace System.Runtime.Remoting.Channels.Ipc de faire du remoting entre plusieurs applications sur le même pc, et ce, bien plus rapidement car n'utilise pas le réseau.

Support SMTP

Grâce aux nouvelles classes System.Net.Mail et System.Net.Mime, il est dorénavant possible d'envoyer un mail depuis n'importe quelle application et ce, en pouvant personnaliser totalement le mail (un ou plusieurs destinataires, mode d'inclusion, ajout de pièce jointes, encodage, etc).

Support du port série

Principalement grâce à la classe SerialPort, il sera maintenant possible de communiquer très facilement avec le port série de l'ordinateur et ainsi mieux contrôler les périphériques qui lui sont attachés.

Bien d'autres nouveautés feront leur apparition mais il ne m'est pas possible de toutes les citer ici.

Conclusion

Pour finir, nous pouvons déjà conclure que le framework 2.0 comporte d'importantes évolutions par rapport au framework 1.1 et nous ne pouvons nous empêcher de féliciter les développeurs du framework 2.0 qui ont travaillé dur pour nous concocter des classes qui faciliteront notre vie professionnelle de développeurs.
A quand l'arrivée d'une prochaine version du framework :)

Liens complémentaires

Spécification du langage C#

Remerciements

Un remerciement tout spécial à **Laurent DARDENNE** pour tout le temps qu'il a passé à m'aider à améliorer cet article

Téléchargements

 **Article au format PDF**