

# Tour d'horizon des nouveautés de C# 4.0

par Jérôme Lambert ([Autres articles](#)) ([Blog](#))

Date de publication : 09/12/2008

Dernière mise à jour :

C'est lors de la Microsoft PDC 2008 (Professional Developer Conference) que Anders Hejlsberg - père spirituel de C# - a dévoilé le voile sur la prochaine version de son langage : C# 4.0.

Au cours de cet article, nous ferons un tour d'horizon des nouveautés qui nous attendent pour cette nouvelle version du langage.

Commentez cet article :

1 - Avant propos.....	3
2 - Introduction.....	3
3 - Le langage C#.....	3
3-1 - Dynamically Typed Objects.....	5
3-1-1 - Mécanisme de résolution à l'exécution.....	8
3-1-2 - Le Dynamic Language Runtime.....	8
3-2 - Optional and Named Parameters.....	9
3-2-1 - Paramètres optionnels.....	9
3-2-2 - Paramètres nommés.....	10
3-3 - Improved COM Interoperability.....	10
3-3-1 - Mapping automatique.....	10
3-3-2 - Paramètres optionnels et paramètres nommés.....	11
3-3-3 - Modificateur ref optionnel.....	11
3-4 - Co- and Contra-Variance.....	12
3-5 - Covariance.....	12
3-6 - Contra variance.....	13
3-7 - Co et Contra variance combinés.....	15
4 - Coévolution des langages C# et VB.NET.....	15
5 - Conclusion.....	16
Remerciements.....	16

## 1 - Avant propos

Les exemples de cet article ont été réalisés avec la CTP (Community Technology Preview) de Microsoft Pre-release Software Visual Studio 2010 et .NET Framework 4.0.

Pour télécharger l'image virtuelle de cette CTP :

<http://www.microsoft.com/downloads/details.aspx?FamilyId=922B4655-93D0-4476-BDA4-94CF5F8D4814&displaylang=en>

La CTP est délivrée dans une image Virtual PC. Il vous faudra donc télécharger Virtual PC 2007 et son Service Pack 1 :

- [Téléchargez Virtual PC 2007](#)
- [Téléchargez le service pack 1 de Virtual PC 2007](#)

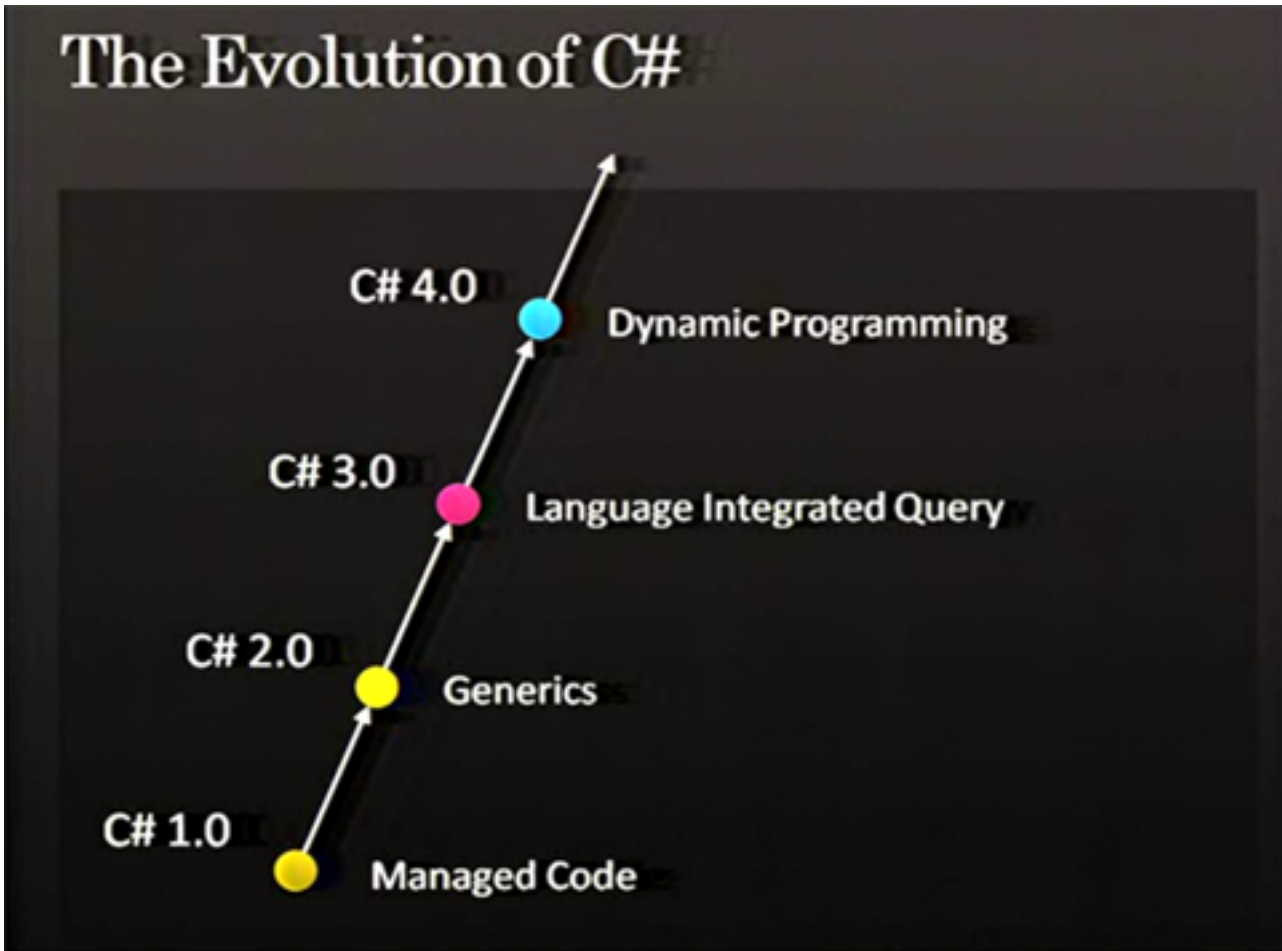
## 2 - Introduction

C'est lors de la Microsoft PDC 2008 que Microsoft a annoncé la mise à disposition de la toute première CTP de Visual Studio 2010 et du .NET Framework 4.0. C'est aussi durant cet événement que le célèbre Anders Hejlsberg, père spirituel de C#, a présenté une session sur le futur C# et je me doute que bon nombre d'entre vous (tout comme moi !) aurait tant aimé y participer.

Au cours de cet article, nous ferons un tour d'horizon sur les nouveautés majeures qu'offre cette préversion du langage C# 4.0 : objets dynamiques, paramètres nommés et optionnels, variance et covariance, . nous verrons tout en détail.

## 3 - Le langage C#

Le langage C# est un langage de programmation orienté objet à typage fort créé par Anders Hejlsberg (le créateur du langage Delphi) au sein de la société Microsoft.



\* Slide provenant de la présentation « The Future of C# » de Anders Hejlsberg lors de la Microsoft PDC 2008

Comme vous pouvez le voir sur l'écran ci-dessus représentant l'évolution de C#, c'est avec C# 1.0 que tout a commencé lorsque le Framework .NET 1.0 est apparu. Très proche de Java, la syntaxe est relativement semblable aux langages C et C++.

Avec C# 2.0 du Framework .NET 2.0, ce fut l'apparition d'une fonctionnalité très attendues par les développeurs et pourtant inexistante dans la version précédente : les génériques. On notera aussi l'apparition des classes partielles, des méthodes anonymes, des types « nullable » et j'en passe tellement la liste serait longue.

Pour C# 3.0 et contrairement à ce qu'on pourrait penser, ce n'est pas le Framework .NET 3.0 qui introduit cette nouvelle version du langage mais bien le Framework .NET 3.5. On notera l'apparition d'une fonctionnalité majeure qui est **Linq** (« *Language Integrated Query* ») dont le but est de permettre de requêter n'importe quel type de source de données de la même façon : Linq to Objects, Linq to XML, Linq to SQL, etc.

Cette nouvelle version du langage apporte aussi son lot de changements au niveau syntaxique rendant possible Linq :

- Variables locales typées implicitement
- Méthodes d'extension
- Expressions lambda
- Initialiseurs d'objets
- Types anonymes

A noter que le code compilé en C# 3.0 est entièrement compatible avec C# 2.0 étant donné que les nouveautés de C# 3.0 sont purement syntaxiques et sont converties en du code C# 2.0 lors de la compilation.

C'est donc à partir de C# 3.0 que Microsoft nous confirme que C# 2.0 est une base assez solide pour que les prochaines versions de C# puisse se reposer dessus.

Depuis quelques jours Concernant C# 4.0, qui sera disponible avec le Framework 4.0 en 2010 dans sa version finale, les deux thèmes majeurs sont :

- **La programmation dynamique**
- **La coévolution des langages**

Mais je m'arrête là pour cette introduction à C#, si vous voulez en savoir plus sur C# 4.0, lisez la suite.

### 3-1 - Dynamically Typed Objects

Une des grandes nouveautés de C# est sans conteste la possibilité de déclarer des instances comme étant dynamiques. Concrètement, vous allez pouvoir :

- Appeler n'importe quelle méthode
- Accéder à un index
- Utiliser un opérateur
- Accéder aux champs et propriétés

Et cela sans qu'il y ait une vérification lors de la compilation ! En fait, c'est lors de l'exécution que tout sera résolu automatiquement.

Pour cela, C# a introduit un nouveau mot clé : **dynamic**. C'est grâce à ce mot clé que l'on va pouvoir déclarer des instances comme étant dynamiques. Mais quel est l'intérêt vous allez me dire ? Par exemple :

- Manipuler plus facilement des objets venant de langage de programmation dynamique comme Python et Ruby (respectivement IronPhyton et IronRuby sous .NET)
- Accéder à des objets COM encore plus facilement
- Simplifier l'utilisation d'objets où la réflexion est la seule alternative
- Manipuler des objets dont la structure change, comme avec des objets HTML DOM

L'objectif est d'unifier l'accès d'objets de différents type peu importe le cas de figure (voir ci-dessus).

Prenons un simple exemple :

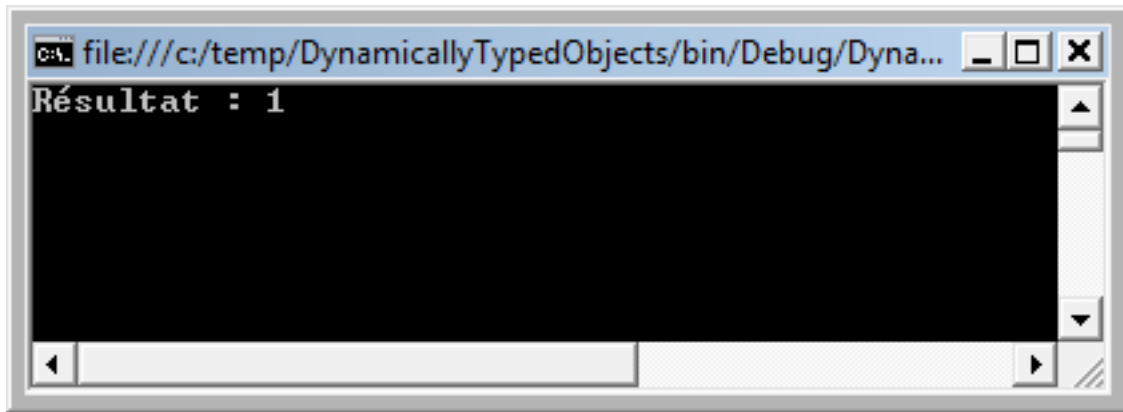
```
dynamic i = 10;
int result = i.CompareTo(5);
Console.WriteLine("Résultat : {0}", result);
```

Vous allez me dire que tout est normal à part l'utilisation du mot clé `dynamic` qui ne change rien à l'histoire, et pourtant détrompez-vous, ce mot clé fait toute la différence. En ayant déclaré la variable `i` (qui devrait être de type `System.Int32`) avec `dynamic`, le compilateur ne sait pas du tout quel est le type réel de la variable « `i` ». C'est pourquoi lorsque je vais écrire la méthode « `CompareTo` », l'intellisense de Visual Studio ne va même pas me proposer la méthode en question dans la liste des méthodes disponibles avec la variable « `i` » (d'ailleurs, il ne va rien me proposer). Logique, il ne connaît pas le type. Par contre, il va me permettre d'écrire ce que je veux et à la compilation, ça ne provoquera ni avertissement, ni erreur.

Comme je l'ai dit, c'est lors de l'exécution que ces opérations vont être résolues. En fait, ma seconde ligne où je fais appel à la méthode « `CompareTo` » peut être traduite comme ceci :

« Dans l'instance de `i`, trouve moi une méthode appelée '`CompareTo`' et qui prend en paramètre un `int` ».

Et le résultat est édifiant !



Comme je l'ai dit, cette programmation dynamique ne s'arrête pas au simple fait d'appeler des méthodes dynamiquement, il y a aussi les index, opérateurs, propriétés, etc qui peuvent en profiter.

```
dynamic d = new List<int>() { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
// Appel de méthode
d.Add(5);
// Accès à une propriété
d.Capacity = 1000;
// Accès à l'index (ne fonctionne pas encore dans la CTP)
d[0] = d[1];
// Utilisation d'opérateur (ne fonctionne pas encore dans la CTP)
int result = d.Capacity + 10;
// Utiliser l'objet dynamique comme un délégué (n'est pas utilisable dans cet exemple)
d(5);
```

Malgré les possibilités offertes dans cette version de C#, tout n'est pas encore possible ::

- Les méthodes d'extensions ne sont pas reconnues étant donné qu'elles font parties d'un contexte statique qui n'est pas encore pris en compte lors de la résolution d'appels dynamiques.
- Les expressions lambda ne peuvent pas être utilisées en paramètres d'une méthode d'instance.

Un dernier exemple amusant que j'ai développé en me basant sur la présentation « The Future of C# » pour tester ce système de programmation dynamique avec C# : j'ai créé deux classes comportant deux propriétés du même nom, à côté de ça, j'ai créé une méthode statique qui prend en paramètre un objet dit dynamique. La suite dans l'exemple qui suit :

```
public static void Main()
{
    ExempleDynamic();

    Console.Read();
}

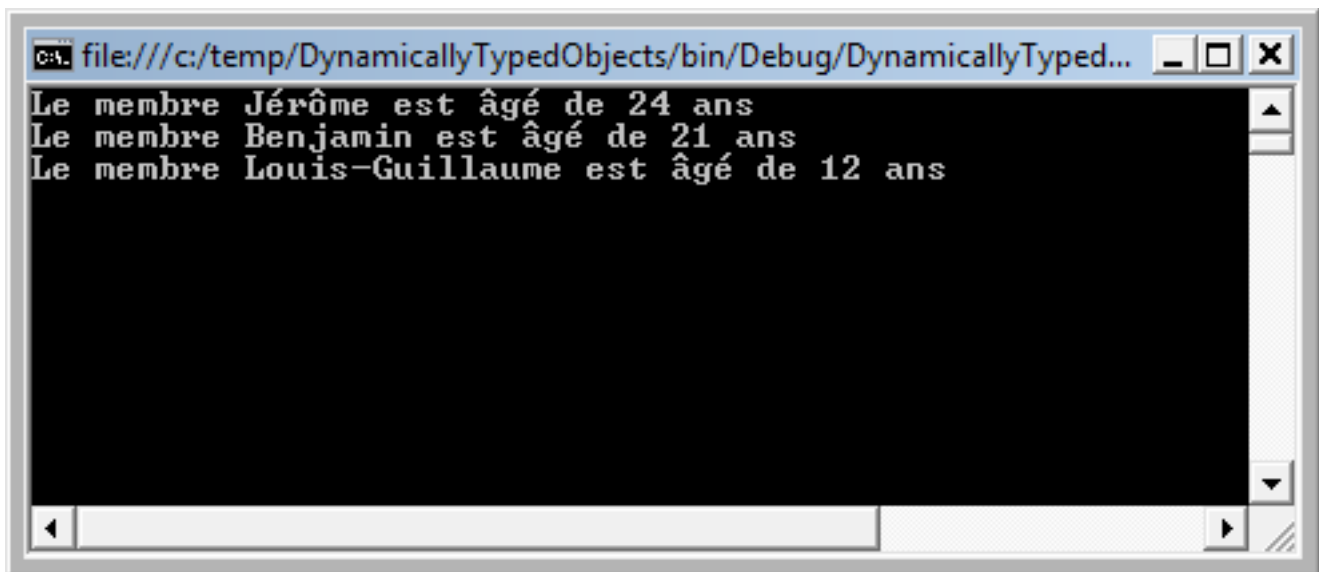
public class ResponsableDvp
{
    public string Nom { get; set; }
    public int Age { get; set; }
    public string Fonction { get; set; }
}

public class MembreDvp
{
    public string Nom { get; set; }
    public int Age { get; set; }
}

public static void ExempleDynamic()
{
    ResponsableDvp membreResponsableDvp = new ResponsableDvp() { Nom = "Jérôme", Age = 24,
    Fonction = "Responsable .NET" };
```

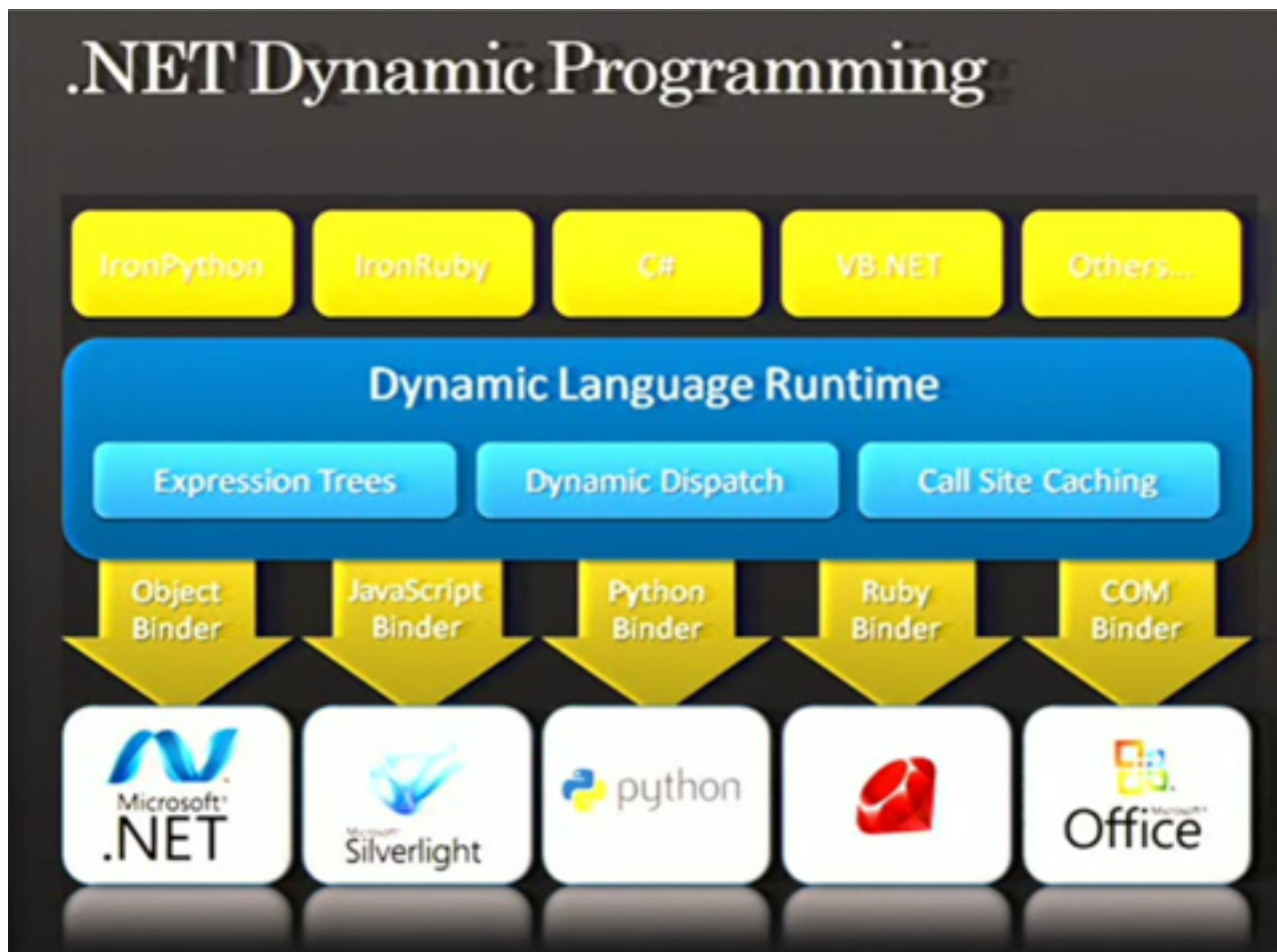
```
MembreDvp membreDvp = new MembreDvp() { Nom = "Benjamin", Age = 21 };  
dynamic membreDvpDynamic = new MembreDvp() { Nom = "Laurent Dardenne", Age = 25 };  
var membreAutre = new { Sexe = 'M', Nom = "Louis-Guillaume", Age = 12 }; // Objet anonyme  
  
AfficherMembre(membreResponsableDvp);  
AfficherMembre(membreDvp);  
AfficherMembre(membreDvpDynamic);  
AfficherMembre(membreAutre);  
}  
  
public static void AfficherMembre(dynamic membre)  
{  
    Console.WriteLine("Le membre " + membre.Nom + " est âgé de " + membre.Age + " ans");  
}
```

Ce qui donne à l'exécution :



Vous remarquerez au passage que j'ai créé en plus un objet anonyme avec des propriétés du même nom que celles qui sont utilisées par ma méthode « AfficherMembre » et ça fonctionne tout aussi bien.

### 3-1-1 - Mécanisme de résolution à l'exécution



\* Slide provenant de la présentation « The Future of C# » de Anders Hejlsberg lors de la Microsoft PDC 2008

Comme vous l'avez compris depuis un moment à présent, la résolution des objets dynamiques se fait au moment de l'exécution mais il faut savoir que cette résolution va être différente selon le type de l'objet dynamique. Il existe en fait 3 types d'objets :

- **Les objets COM**, l'opération est dispatchée dynamiquement à l'interface IDispatch de l'objet COM en question.
- **Les objets dynamiques**, ce sont des objets qui implémentent la nouvelle interface IDynamicObject et dans ce cas, c'est l'objet lui-même qui va s'occuper de résoudre l'opération. En implémentant cette interface dans nos classes, on va pouvoir personnaliser complètement leur comportement lors de leur utilisation sous forme d'objets dynamique.
- **Les objets simples**, c'est tout simplement les objets .NET dont les opérations seront transformées en appels utilisant la réflexion avec utilisation d'un « runtime binder » comme vous pouvez le voir sur le précédent schéma.

### 3-1-2 - Le Dynamic Language Runtime

Rien de tout cela ne serait possible sans la « Dynamic Language Runtime » qui est une nouvelle API du Framework .NET 4.0 permettant les appels dynamique.

Cette API est en fait une surcouche de la CLR (Common Language Runtime) et permet de faciliter l'implémentation et l'interopérabilité des langages dynamiques avec la plateforme.NET, les plus connus sont IronPython et IronRuby.

Pour en savoir plus sur la DLR, je vous invite à consulter la fiche Wikipedia : [http://en.wikipedia.org/wiki/Dynamic\\_Language\\_Runtime](http://en.wikipedia.org/wiki/Dynamic_Language_Runtime)

## 3-2 - Optional and Named Parameters

Une fonctionnalité qui existait déjà en Visual Basic et qui se faisait attendre pour certains développeurs C#, ce sont les paramètres nommés et les paramètres optionnels.

Si on prend le cas des API de l'automation Office, bon nombre de méthodes regorgent de méthodes composées de paramètres nommés et de paramètres optionnels. C# ne supportant pas ces fonctionnalités, on était à chaque fois obligé de passer tous les paramètres et pour ceux qui ne nous intéressaient pas, on donnait comme valeur « null » ou une valeur par défaut. On arrivait dans la plupart des cas à des appels illisibles.

Avec C# 4.0, cela est définitivement fini.

### 3-2-1 - Paramètres optionnels

Pour déclarer un paramètre d'une méthode comme optionnel, il suffit tout simplement de lui affecter une valeur par défaut :

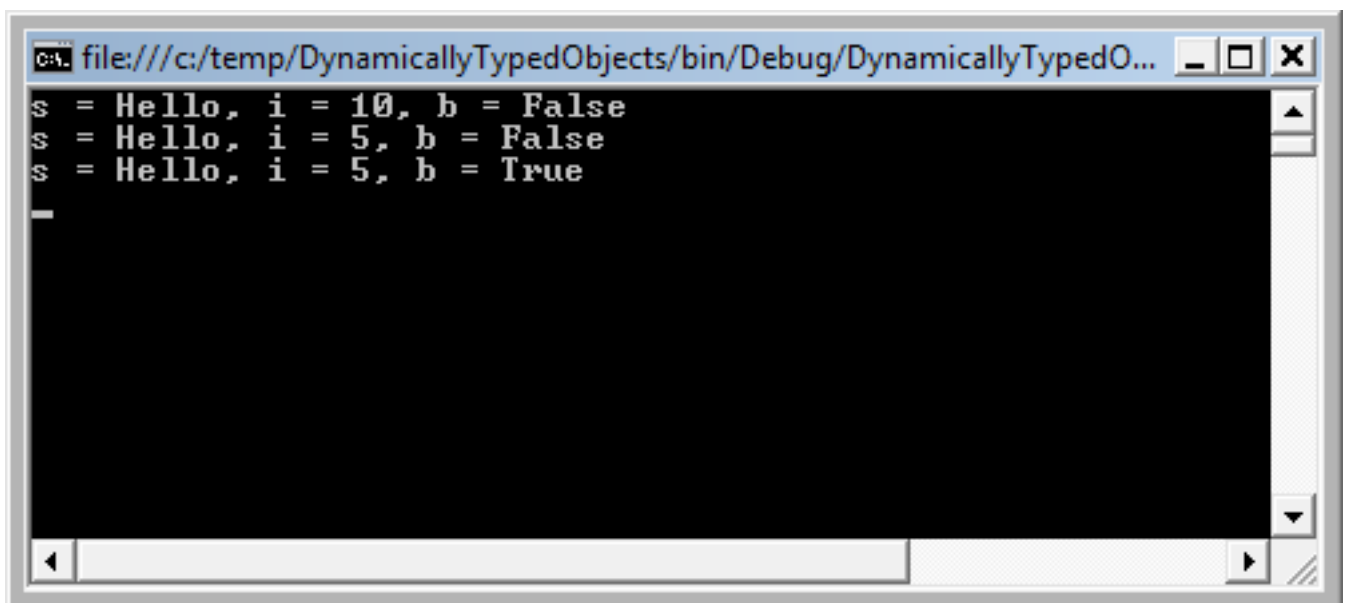
```
public static void MaMethode(string s, int i = 10, bool b = false)
{
    Console.WriteLine("s = {0}, i = {1}, b = {2}", s, i, b);
}
```

Et pour appeler cette méthode, je pourrais omettre le second ou le troisième paramètres si je le désire :

```
MaMethode("Hello");
MaMethode("Hello", 5);
MaMethode("Hello", 5, true);

// MaMethode("Hello", false); // Interdit
```

Ce qui donnera à l'exécution :



```
file:///c:/temp/DynamicallyTypedObjects/bin/Debug/DynamicallyTypedO...
s = Hello, i = 10, b = False
s = Hello, i = 5, b = False
s = Hello, i = 5, b = True
```

Au passage, vous remarquez que je ne peux pas omettre le second paramètre tout en précisant le troisième pour la bonne et simple raison que les paramètres qui sont laissés optionnels doivent tous se trouver après le dernier paramètre spécifié.

### 3-2-2 - Paramètres nommés

Pour spécifier un paramètre nommé, il suffit lors de l'appel de la méthode de préciser le nom du paramètre suivi de « : » puis la valeur du paramètre. Ainsi, on va pouvoir spécifier les paramètres dans l'ordre que l'on désire.

```
MaMethode("Hello", 5, b: true);  
MaMethode(s: "Hello", i: 5, b: true);  
MaMethode(b: true, i: 5, s: "Hello");  
  
// MaMethode(s: "Hello", 5, b: true); // Interdit
```

Dans le cas où on désire utiliser des paramètres non-nommés avec des paramètres nommés, ces derniers doivent toujours se trouver en dernières positions.

A savoir aussi que les paramètres optionnels et les paramètres nommés peuvent non seulement être utilisés avec les méthodes mais aussi avec les constructeurs et les indexeurs.

### 3-3 - Improved COM Interoperability

Précédemment, nous avons parlé de deux nouvelles fonctionnalités très importantes de C# 4.0 et qui sont le système des types dynamiques, les paramètres nommés et les paramètres optionnels. Ces fonctionnalités ne sont pas anodines au niveau de l'interopérabilité COM, comme par exemple avec l'automation Office. Nous allons voir en quoi ces nouvelles fonctionnalités vont aider à améliorer le développement avec des objets COM.

#### 3-3-1 - Mapping automatique

Prenons un exemple tout simple qui est la création d'une feuille Excel dans laquelle on va écrire du texte dans une cellule.

```
var excel = new Microsoft.Office.Interop.Excel.Application();  
  
(Microsoft.Office.Interop.Excel.Range)excel.Cells[1, 1].Value2 = "Ma feuille Excel";
```

Le problème est qu'on est sans cesse obligé de convertir les objets. Dans l'exemple précédent l'accès à l'indexeur renvoie un objet mais moi en tant que développeur, je sais que l'objet renvoyé est évidemment de type « Excel.Range ». Et bien aujourd'hui, c'est fini ! Avec C# 4.0, vous pourrez dorénavant écrire :

```
var excel = new Microsoft.Office.Interop.Excel.Application();  
  
excel.Cells[1, 1].Value2 = "Ma feuille Excel";
```

En fait, nous avons déjà parlé de cette nouvelle fonctionnalité, c'est le principe de la programmation dynamique. Le compilateur considère que « excel.Cells[ ] » renvoie un type dynamique, donc libre à vous d'accéder aux propriétés et méthodes que vous désirez et ce n'est que lors de l'exécution que la résolution sera effectuée :

- J'ai un type dynamique - « *excel.Cells[1, 1]* »
- Auquel je dois vérifier s'il existe une propriété nommée « Value2 » de type string avec un accesseur « set » - « *Value2 = "Ma feuille Excel"* »

### 3-3-2 - Paramètres optionnels et paramètres nommés

Si vous avez l'habitude de développer en utilisant des objets COM, vous vous êtes sans conteste heurté aux paramètres de méthodes qui ne vous intéresseront jamais et pourtant, vous êtes obligé de passer une valeur par l'intermédiaire de « System.Type.Missing ». Si on prend le code permettant de créer une nouvelle feuille à notre classeur Excel sans vouloir spécifier sa position, on écrira tout simplement :

```
var excel = new Microsoft.Office.Interop.Excel.Application();

var newSheet = excel.Sheets.Add(System.Type.Missing,
                                System.Type.Missing,
                                System.Type.Missing,
                                System.Type.Missing);
```

On remarquera que la méthode « Add » requiert quatre paramètres optionnels qu'il est obligatoire de préciser mêmes si ces derniers ne nous intéressent pas.

Avec C# 4.0, le code deviendra tout simplement :

```
var excel = new Microsoft.Office.Interop.Excel.Application();

var newSheet = excel.Sheets.Add();
```

Mieux encore, si je désire créer 5 feuilles Excel en même temps, il suffit que je renseigne le troisième paramètre optionnel « Count » représentant le nombre de feuille Excel que l'on désire créer. Ici je vais pouvoir tout simplement nommer mon paramètre comme on l'a déjà vu précédemment :

```
var excel = new Microsoft.Office.Interop.Excel.Application();

var newSheet = excel.Sheets.Add(Count: 5);
```

### 3-3-3 - Modificateur ref optionnel

Une autre petite nouveauté est l'utilisation des paramètres passés par référence. De nombreuses méthodes d'objets COM nécessitent de passer les paramètres par référence alors que dans la plupart des cas, cela devrait être de simples paramètres valeurs. Tout cela oblige le développeur à créer une variable locale pour ensuite la passer en paramètre de la méthode précédé du modificateur « ref » permettant de passer une variable de type valeur par référence. Ce qui donne par exemple :

```
var word = new Microsoft.Office.Interop.Word.Application();

object missing = System.Type.Missing;
object visible = true;
word.Documents.Add(ref missing, ref missing, ref missing, ref visible);
```

Avec C# 4.0, il ne sera plus nécessaire d'utiliser « ref » et ainsi, on pourra écrire :

```
var word = new Microsoft.Office.Interop.Word.Application();

word.Documents.Add(System.Type.Missing, System.Type.Missing, System.Type.Missing, true);
```

En fait, c'est le compilateur qui s'occupera de créer pour vous une variable locale pour la passer ensuite par référence.



*Cette nouveauté, bien que décrite dans le document « New features in CSharp 4.0 », ne semble pas encore être opérationnelle dans la CTP actuelle de Visual Studio 2010.*

## 3-4 - Co- and Contra-Variance

Avant de commencer à vous parler ce que nous réserve C# 4.0 en terme de Co et Contra variance, il est bon de faire un rappel sur ce qu'est la variance. Moi-même quand j'ai lu ça, je ne me suis pas tout de suite rappelé ce que cela signifiait et pourtant, c'est quelque chose d'omniprésent que tout développeur connaît sans pour autant connaître le terme caché derrière.

Prenons le cas d'une liste générique de chaîne de caractères :

```
List<string> myStrings = new List<string>();
```

L'avantage d'une telle liste est que le compilateur vérifiera que tous les accès à ma liste générique ne concernent que des types string. Ainsi, si je désire ajouter un int, cela sera impossible dès la compilation.

```
myStrings.Add(« Hello World »); // OK  
myStrings.Add(2008); // Erreur de compilation
```

Mais pas contre, qui n'a jamais essayé une conversion du même genre que celle qui suit ?

```
IEnumerable<object> myObjects = myStrings;
```

Pour ceux qui ont déjà essayé, cela ne fonctionne pas car on est dans ce qu'on appelle la variance de générique, ce qui n'est pas pris en charge par C#. Il est vrai qu'un string dérive du type object mais dans le cas de nos listes, cela n'est pas possible.

Avec C# 4.0, vous allez voir comment ce problème a été résolu.

## 3-5 - Covariance

Avec C# 4.0, une nouvelle signification a été donnée au mot clé « out » dans le but de permettre la covariance avec les génériques.

```
interface IMembres<out T> { }  
class Membres<T> : IMembres<T> { }  
class Modérateur { }  
class Responsable : Modérateur { }  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        IMembres<Responsable> responsables = new Membres<Responsable>();  
        IMembres<Modérateur> modérateurs = responsables;  
        IEnumerable<object> membres = responsables;  
    }  
}
```

Grâce à l'utilisation du mot clé « out » dans la déclaration de l'interface « IMembres », il est à présent possible de convertir un « IMembres<Responsable> » en « IMembres<Modérateur> » étant donné que la classe « Responsable » dérive de « Modérateur ». Ce système n'est donc utile que dans la position de sortie (output) des variables.

Cet exemple est similaire à l'interface « IEnumerable<T> » et pour cause, à partir du Framework .NET 4.0, cette interface sera étendue de la sorte :

```
public interface IEnumerable<out T> : IEnumerable  
{  
    IEnumerator<T> GetEnumerator();  
}
```

```
public interface IEnumerator<out T> : IEnumerator
{
    bool MoveNext();
    T Current { get; }
}
```

Ainsi, on pourra envisager la conversion suivante :

```
List<string> mesStrings = new List<string>();
IEnumerable<object> mesObjects = mesStrings;
```

Contrairement à ce que j'ai pu lire sur certains blogs, ce système de covariance sur les génériques ne s'applique que sur les interfaces et délégués et pas (ou pas encore) aux classes et méthodes.

Une autre chose importante à préciser, la conversion dite de « boxing », c'est-à-dire la conversion d'un type valeur (int, float, double, char, les structures, etc), n'est pas autorisée. Seule la conversion de référence est permise pour l'instant.

```
IEnumerable<object> mesObjects = IEnumerable<string>; // Conversion de référence autorisée
IEnumerable<object> mesObjects = IEnumerable<int>; // Boxing interdit
```

### 3-6 - Contra variance

A l'inverse du mot clé out, nous avons le mot clé « in » qui va autoriser la contra variance au niveau des génériques. Ici on va pouvoir agir sur les paramètres d'entrées. Pour l'exemple suivant, je vais reprendre celui que propose l'équipe C# dans son document « New features in CSharp 4.0 », il me semble assez explicite :

```
public interface IComparer<in T>
{
    int Compare(T left, T right);
}
```

L'interface « IComparer » autorise la contra variance de « T » grâce au modificateur « in », ce qui signifie que si vous avez un « IComparer<object> », vous pourrez l'utiliser comme un « IComparer<string> ».

Un exemple d'utilisation ne sera pas de trop : prenons le cas de deux classes "MembreDvp" qui contient une propriété "Nom" et "ResponsableDvp" qui dérive de la classe "MembreDvp". A ça, je vais créer une classe "myComparer" qui implémente l'interface "IComparer<MembreDvp>" pour faire une comparaison sur base de la propriété "Nom". Ainsi, il va nous être possible de trier une liste de "MembreDvp" grâce à la propriété "Sort" de la classe "List<T>" qui prend en paramètre un IComparer. Mais imaginons que nous avons une liste de "ResponsableDvp" et que nous désirons la trier sur base du nom (propriété qui provient de la classe "MembreDvp" pour rappel), jusqu'à C# 3.0, il est impossible de réutiliser notre classe "myComparer" et la seule solution est de créer un autre classe qui implémente un "IComparer<ResponsableDvp>", admettez que dans notre cas, c'est du travail en trop et c'est tout à fait légitime. Rassurez-vous, avec C# 4.0, tout cela sera définitivement terminé comme démontré dans l'exemple suivant :

```
public class MembreDvp
{
    public string Nom { get; set; }
}

public class ResponsableDvp : MembreDvp
{ }

class myComparer : IComparer<MembreDvp>
{
    public int Compare(MembreDvp x, MembreDvp y)
    {
        if (x == null && y == null)
        {
            return 0;
        }
        else if (x == null)

```

```

    {
        return -1;
    }
    else if (y == null)
    {
        return 1;
    }
    else
    {
        return x.Nom.ToLower().CompareTo(y.Nom.ToLower());
    }
}

class Program
{
    static void Main(string[] args)
    {
        myComparer comparer = new myComparer();

        List<MembreDvp> membresDvp = new List<MembreDvp>();
        membresDvp.Add(new MembreDvp() { Nom = "Thomas Lebrun" });
        membresDvp.Add(new MembreDvp() { Nom = "Skyounet" });
        membresDvp.Add(new MembreDvp() { Nom = "the_badger_man" });
        membresDvp.Add(new MembreDvp() { Nom = "Aspic" });
        membresDvp.Add(new MembreDvp() { Nom = "tomlev" });

        membresDvp.Sort(comparer);

        Console.WriteLine("Tri des membres Dvp");
        Console.WriteLine("*****");
        foreach (var membre in membresDvp)
            Console.WriteLine(membre.Nom);

        List<ResponsableDvp> responsableDvp = new List<ResponsableDvp>();
        responsableDvp.Add(new ResponsableDvp() { Nom = "Louis-Guillaume Morand" });
        responsableDvp.Add(new ResponsableDvp() { Nom = "Jérôme Lambert" });
        responsableDvp.Add(new ResponsableDvp() { Nom = "Stéphane Eyskens" });
        responsableDvp.Add(new ResponsableDvp() { Nom = "Ricky81" });
        responsableDvp.Add(new ResponsableDvp() { Nom = "Fleur-Anne Blain" });

        responsableDvp.Sort(comparer);

        Console.WriteLine();
        Console.WriteLine("Tri des responsables Dvp");
        Console.WriteLine("*****");
        foreach (var membre in responsableDvp)
            Console.WriteLine(membre.Nom);

        Console.Read();
    }
}

```

Et le résultat :

```
e:///C:/Users/jerome/Documents/Visual Studio 2008/Projects/IComparer/IComparer/bin/Debug/ICompare...
```

```
des membres Dup
```

```
*****
```

```
c  
unet  
badger_man  
as Lebrun  
ev
```

```
des responsables Dup
```

```
*****
```

```
r-Anne Blain  
me Lambert  
s-Guillaume Morand  
y81  
hane Eyskens
```

### 3-7 - Co et Contra variance combinés

Il est tout à fait possible d'avoir un type générique qui autorise les modificateurs « out » et « in » en même temps. Prenons l'exemple du délégué « Func » qui propose la signature suivante :

```
public delegate TResult Func<T, TResult>(T arg)
```

Ici « T » est un paramètre d'entrée et « TResult » le paramètre de sortie. Ça vous rappelle quelque chose ?! Paramètre de sortie, donc utilisation de « out », paramètre d'entrée, utilisation de « in ». Ce qui donnerait comme signature :

```
public delegate TResult Func<in T, out TResult>(T arg)
```

Ainsi pour une même expression lambda, on pourrait utiliser les déclarations de délégués suivantes :

```
Func<object, string> func1 = s => s.ToString();  
Func<string, object> func2 = s => s.ToString();
```

### 4 - Coévolution des langages C# et VB.NET

Une autre des grandes nouvelles est la coévolution des langages du Framework .NET, et plus particulièrement C# et Visual Basic. Rassurez-vous, cette coévolution ne concerne que les fonctionnalités, ce qui signifie que lorsque l'équipe Visual Basic mettra en place une nouvelle fonctionnalité pour son langage, cette fonctionnalité sera implémentée par l'équipe C# aussi, et vice-versa.

**Ceci sera d'application pour C# 4.0 et VB.NET 10** et pour ceux qui désirent revoir l'annonce faites par les responsables des équipes C# et VB.NET lors de la Microsoft PDC, je vous invite à aller voir les webcasts suivantes :


- **The Future of C#**, présenté par *Anders Hejlsberg*
- **Future Directions for Visual Basic**, présenté par *Paul Vick* et *Lucian Wischik*

Je vous invite au passage à réagir sur cette nouvelle via notre forum : « **La coévolution des langages Visual Basic et C# est maintenant officielle** »

## 5 - Conclusion

Comme vous avez pu le constater tout au long de cet article, l'équipe en charge du langage C# ne cesse de nous surprendre avec des fonctionnalités à chaque fois plus époustouflantes. D'ailleurs, pour ceux qui n'ont pas encore visionné la session « **The Future of C#** » présenté par Anders Hejlsberg, pensez à jeter un oeil, ça vaut le détour.

N'oubliez pas non plus que tout ceci n'est qu'en CTP, on en est donc même pas au stade de version alpha mais tout cela nous permet d'avoir un aperçu de quoi C# sera fait dans 2 ans même si pas mal de choses évolueront d'ici là.

Pour plus d'informations concernant la CTP de C# 4.0, une seule adresse :  **C# Future sur MSDN Code Gallery**

## Remerciements

Je tiens à remercier toute l'équipe .NET pour son aide précieuse et plus particulièrement SpiceGuid et Laurent Dardenne pour leurs remarques et conseils avisés. Petit clin d'œil à notre Badger pour ses compétences avec les addin Office : merci à toi pour le temps gagné !