



Dotnet France
Technologies Sharepoint, SQL Server & .NET

Association Dotnet France

Les nouveautés du langage C# 4.0

Version 1.1



James RAVAILLE

<http://blogs.dotnet-france.com/jamesr>

Sommaire

1	Introduction.....	3
1.1	Présentation	3
1.2	Pré-requis	3
2	Les paramètres nommés et optionnels.....	4
2.1	Présentation	4
2.1.1	Les paramètres optionnels.....	4
2.1.2	Les paramètres nommés	4
2.1.3	Résolution des conflits	5
3	Le typage dynamique	6
3.1	Présentation	6
3.2	Exemples.....	6
4	La co-variance et la contre-variance	10
4.1	Co-variance et contre-variance sur les délégués	11
4.1.1	La co-variance et les interfaces génériques	12
4.1.2	La contre-variance et les interfaces génériques.....	14
5	Conclusion	17

1 Introduction

1.1 Présentation

Au début de l'année 2010, Microsoft proposera Visual Studio 10, la version 4.0 du Framework .NET, ainsi qu'une nouvelle version du langage C# (version 4.0). 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 paramètres nommés et optionnels.
- Les instructions dynamiques.
- La co-variance et la contre-variance.

Aussi, ce cours est basé sur la version Beta 1 du Framework .NET 4.0 et Visual Studio 2010 Beta 1.

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.
- Les nouveautés du langage C# 3.0.

2 Les paramètres nommés et optionnels

2.1 Présentation

Le langage C# 4.0 autorise les paramètres nommés et les paramètres optionnels. Ils peuvent être utilisés lors de la définition de méthodes, constructeurs et indexeurs. Soit la méthode suivante, permettant d'afficher la somme de trois nombres passés en paramètre :

```
// C#  
  
public void Calculer(int x, int y = 15, int z = 67)  
{  
    MessageBox.Show((x + y + z).ToString());  
}
```

2.1.1 Les paramètres optionnels

La méthode *Calculer* accepte trois paramètres. Les deux derniers paramètres acceptent une valeur par défaut : il s'agit de paramètres optionnels (aucun mot clé n'est nécessaire). Ces valeurs seront utilisées si la méthode est appelée, sans spécifier de valeur pour ces paramètres.

Une règle d'or à respecter : dans une méthode, si un paramètre est défini comme optionnel, alors tous les autres paramètres doivent aussi être définis comme optionnels.

Pour appeler cette méthode, il est possible d'utiliser l'une des instructions suivantes :

```
// C#  
  
this.Calculer(5);  
this.Calculer(5, 42);  
this.Calculer(5, 42, 11);  
this.Calculer(5, , 11);
```

La première instruction exécute la méthode *Calculer* avec les valeurs respectives 5, 15 et 67. Le nombre 87 est alors affiché.

La seconde instruction exécute la méthode *Calculer* avec les valeurs respectives 5, 42 et 67. Le nombre 114 est alors affiché.

La troisième instruction exécute la méthode *Calculer* avec les valeurs respectives 5, 42 et 11. Le nombre 58 est alors affiché.

La dernière instruction exécute la méthode *Calculer* avec les valeurs respectives 5, 15 et 11. Le nombre 31 est alors affiché.

2.1.2 Les paramètres nommés

Pour appeler cette méthode, il est possible d'utiliser les paramètres nommés. Leur utilisation permet d'appeler une méthode, en précisant pour un paramètre son nom et une valeur, séparé avec le caractère « : ». Leur utilisation permet de ne pas respecter l'ordre des paramètres.

Voici quelques exemples d'appels de la méthode *Calculer* :

```
// C#  
  
this.Calculer(5, z:12);  
this.Calculer(z:43, y:47, x:3);
```

La première instruction exécute la méthode *Calculer* avec les valeurs respectives suivantes : 5, 15 et 12. Le nombre 32 est alors affiché.

La seconde instruction exécute la méthode *Calculer* avec les valeurs respectives suivantes : 3, 47 et 43. Le nombre 93 est alors affiché.

2.1.3 Résolution des conflits

Soit les méthodes surchargées suivantes :

```
// C#  
  
public void ExecuterTraitement(string s, int i = 1) {}  
public void ExecuterTraitement(object o) {}  
public void ExecuterTraitement(int i, string s = "Actif") {}  
public void ExecuterTraitement(int i) {}
```

Lorsque nous écrivons l'instruction suivante :

```
// C#  
  
this.ExecuterTraitement(15);
```

Quelle méthode est appelée ? Dans un premier temps, en observant les signatures des méthodes, toutes peuvent correspondre. Dans cette instruction accepte un seul paramètre de type *int*. Cette information réduit notre choix aux deux dernières méthodes. Parmi ces deux méthodes, la méthode sans le paramètre optionnel est prioritaire et sera donc exécutée.

De manière générale, la méthode exécutée est la méthode dont la signature correspond le mieux aux paramètres passés, en s'appuyant sur le nombre de paramètres et leurs types.

3 Le typage dynamique

3.1 Présentation

Le typage dynamique des objets permet de créer des instructions dynamiques. Ces instructions permettent déclarer des variables locales, des paramètres de méthodes ou des attributs de classe, qui seront typés lors de l'exécution de l'application (on parle dans ce cas de liaisons tardives). Elles peuvent aussi être utilisées comme type de retour de méthodes.

Le typage dynamique exploite une nouvelle fonctionnalité de la CLR, appelée la DLR (**D**ynamic **L**anguage **R**untime). Cette fonctionnalité permet d'exécuter dans la CLR des langages dynamiques tels que le langage IronPython (« Python » pour .NET).

Une fois déclarée dynamiquement, il est possible d'utiliser ces membres, sans qu'aucune vérification ne soit effectuée par le compilateur.

Les intérêts des instructions dynamiques sont les suivants :

- Utiliser plus facilement des objets venant de langage de programmation dynamique tels que les langages *IronPhyton* et *IronRuby*.
- Faciliter l'accès et l'utilisation d'objets COM.
- Proposer une alternative à la réflexion (mécanismes d'introspection de code MSIL, ...).

Toutefois, attention à l'utilisation des instructions dynamiques. Utilisées à mauvais escient, elles rendront les applications moins robustes lors de leur exécution, en provoquant des erreurs uniquement lors de l'exécution (le compilateur ne connaît pas le type des objets manipulés). Dans des cas très précis, elles permettent de simplifier le développement.

Pour mettre en œuvre le typage dynamique, nous utiliserons le mot clé *dynamic*.

3.2 Exemples

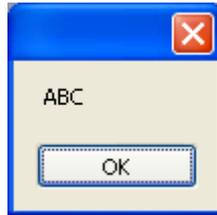
Soit le bloc d'instructions ci-dessous, que nous pouvons exécuter dans une application Windows Forms :

```
// C#  
  
dynamic var1 = "abc";  
string VAR1 = var1.ToUpper();  
MessageBox.Show(VAR1);
```

La première instruction permet de déclarer une variable nommée *var1*, et de la typer (lors de l'exécution) en chaîne de caractères (type *string*), ce type étant déduit de la valeur d'affectation "abc".

La seconde instruction permet d'appliquer (dynamiquement) la méthode *ToUpper* à cette variable, de manière à transformer sa valeur en chaîne de caractères majuscules.

La troisième instruction permet d'afficher dans une boîte de message le contenu de la variable *VAR1*.



Voici un second exemple. Soit les deux classes suivantes :

```
// C#  
  
public class Personne  
{  
    public string Nom { get; set; }  
    public string Prenom { get; set; }  
  
    public Personne(string aNom, string aPrenom)  
    {  
        this.Nom = aNom;  
        this.Prenom = aPrenom;  
    }  
  
    public string Deplacer()  
    {  
        // Bloc d'instructions.  
        // ...  
  
        return this.Nom + " " + this.Prenom + " est arrivé à  
destination";  
    }  
}  
  
public class Animal  
{  
    public string Nom { get; set; }  
    public string Race { get; set; }  
  
    public Animal(string aNom, string aRace)  
    {  
        this.Nom = aNom;  
        this.Race = aRace;  
    }  
  
    public Point Deplacer()  
    {  
        // Bloc d'instructions.  
        // ...  
  
        return new Point(20, 50);  
    }  
}
```

Ces deux classes possèdent un accesseur *Nom* (vous remarquerez l'utilisation des accesseurs simplifiés, cf: *cours sur les nouveautés du langage C# 3.0*) en commun, ainsi qu'une méthode nommée *Deplacer*.

Voici une autre méthode ayant un paramètre défini dynamiquement, appliquant une méthode *Deplacer()* à cet objet, et affiche les informations retournées par cette méthode :

```
// C#  
  
public void ChangerPosition(dynamic aObjet)  
{  
    MessageBox.Show(aObjet.Deplacer().ToString());  
}
```

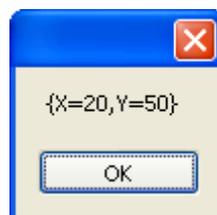
Le bloc d'instructions crée une instance de la classe *Personne*, une autre de la classe *Animal*, et exécute deux fois la méthode *ChangerPosition* en passant successivement ces objets en paramètre :

```
// C#  
  
private void button1_Click(object sender, EventArgs e)  
{  
    Personne oPersonne = new Personne("RAVAILLE", "James");  
    Animal oAnimal = new Animal("Médor", "Chien");  
  
    this.ChangerPosition(oPersonne);  
    this.ChangerPosition(oAnimal);  
}
```

L'instruction *this.ChangerPosition(oPersonne);* affiche le message suivant :



L'instruction *this.ChangerPosition(oAnimal);* affiche le message suivant :



Ces deux exemples mettent en évidence que la méthode *Deplacer* est bien appliquée dynamiquement à un objet, dont le type de données n'est pas explicitement défini lors de la compilation, mais lors de l'exécution. Elle est appliquée sur un objet, quelque soit son type, tant que cette méthode est définie au sein de la classe ou héritée d'une classe de base. Cependant :

- Il n'est pas possible d'appliquer de cette manière les méthodes d'extension (cf : *cours sur les nouveautés du langage C# 3.0*).

- Il n'est pas possible d'utiliser les expressions lambda (*cf* : *cours sur les nouveautés du langage C# 3.0*) comme paramètre des méthodes.

Voici une méthode nommée *Embaucher*, acceptant un paramètre défini dynamiquement, et affichant le *Nom* et le *Prenom* de l'objet passé en paramètre :

```
// C#  
  
public void Embaucher(dynamic aPersonne)  
{  
    MessageBox.Show(aPersonne.Nom + " " + aPersonne.Prenom + " a été  
    embauché le " + DateTime.Now.ToShortDateString());  
}
```

Comme nous l'avons vu, il est alors possible d'appeler cette méthode en passant un objet créé à partir de n'importe quelle classe exposant des propriétés *Nom* et *Prenom*. Mais cet objet peut aussi avoir été créé à partir d'un type anonyme (*cf* : *cours sur les nouveautés du langage C# 3.0*) :

```
// C#  
  
private void button1_Click(object sender, EventArgs e)  
{  
    Personne oPersonne = new Personne("RAVILLE", "James");  
    var oPersonne1 = new { Nom = "VERGNAULT", Prenom = "Bertrand" };  
  
    this.Embaucher(oPersonne);  
    this.Embaucher(oPersonne1);  
}
```

L'instruction *this.Embaucher(oPersonne)*; affiche le message suivant :



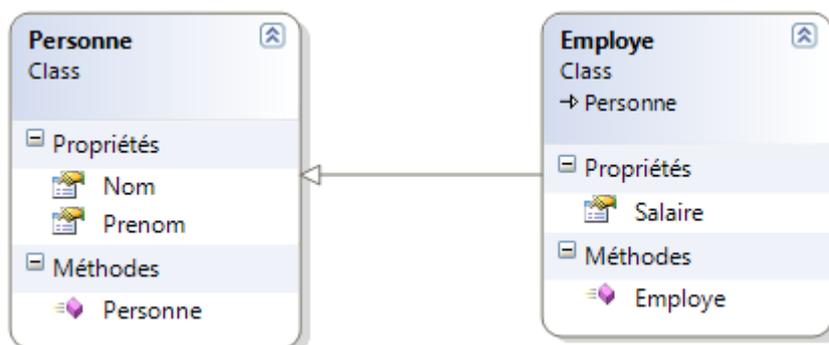
L'instruction *this.Embaucher(oPersonne1)*; affiche le message suivant :



4 La co-variance et la contre-variance

La co-variance et la contre-variance n'est pas un nouveau concept. En C# 2.0, il était possible de les appliquer sur des délégués. Le langage C# 4.0 étend la co-variance et la contre-variance aux interfaces génériques.

Pour illustrer en quoi consiste la co-variance et la contre-variance, nous allons nous baser sur le diagramme de classes suivant :



La classe *Personne* est caractérisée par un nom et un prénom. La classe *Employe* spécialise la classe *Personne*, et définit un attribut supplémentaire : le salaire.

Voici l'implémentation de ces deux classes :

```
// C#

public class Personne
{
    public string Nom { get; set; }
    public string Prenom { get; set; }

    public Personne(string aNom, string aPrenom)
    {
        this.Nom = aNom;
        this.Prenom = aPrenom;
    }
}

public class Employe : Personne
{
    public double Salaire { get; set; }

    public Employe(string aNom, string aPrenom, double aSalaire) :
base(aNom, aPrenom)
    {
        this.Salaire = aSalaire;
    }
}
```

4.1 Co-variance et contre-variance sur les délégués

Nous allons commencer par expliciter les notions de co-variance et de contre-variance sur les délégués, notions déjà existantes dans le langage C# 2.0.

Dans l'exemple présenté ci-dessus, la classe *Employe* dérive de la classe *Personne*. En C# 2.0, on peut alors écrire le bloc de code suivant :

```
// C#

private delegate Personne PersonneHandler(Employe aEmploye);

private Employe Licencier(Employe aEmploye)
{
    return null;
}

private Personne Embaucher(Personne aPersonne)
{
    return null;
}

private void DemoVariance(object sender, EventArgs e)
{
    PersonneHandler oDelegue1 = Licencier;
    PersonneHandler oDelegue2 = Embaucher;
}
```

Détaillons ce bloc de code :

- Dans un premier temps, nous déclarons un délégué nommé *PersonneHandler*. Pour rappel, il s'agit d'un type de données permettant de créer des objets pointant vers une méthode. Ce délégué permet donc de créer des objets pointant vers une méthode, acceptant un employé en paramètre et retournant une personne.
- Puis nous déclarons deux méthodes ayant des signatures différentes. La méthode *Embaucher* accepte une personne en paramètre et retourne une personne. La méthode *Licencier* accepte un employé en paramètre et retourne aussi une personne.
- Dans la méthode *DemoVariance* :
 - o Nous déclarons une instance du délégué nommée *oDelegue1* pointant vers la méthode *Licencier*, bien que cette méthode ne respecte pas strictement la signature du délégué : elle retourne un objet de type *Employe* au lieu de *Personne*. L'utilisation de la **co-variance** permet l'écriture de cette instruction.
 - o Puis nous déclarons une autre instance du délégué nommée *oDelegue2* pointant vers la méthode *Embaucher*, bien que cette méthode ne respecte pas strictement la signature du délégué : elle accepte en paramètre de type *Personne* et non *Employe*. L'utilisation de la **contre-variance** permet l'écriture de cette instruction.

L'utilisation de la co-variance et de la contre-variance permettent aussi d'écrire ces instructions, n'étant qu'une évolution des délégués (toujours valable à partir de C# 2.0) :

```
// C#

Func<Employe, Personne> oFunc1 = Embaucher;
Func<Employe, Personne> oFunc2 = Licencier;
```

L'objet *oFunc1* pointe vers la méthode *Embaucher*. L'objet *oFunc2* pointe vers la méthode *Licencier*.

4.1.1 La co-variance et les interfaces génériques

Dans le Framework .NET 4.0, les interfaces génériques *IEnumerable* et *IEnumerator* sont définies de la manière suivante :

```
// C#  
  
public interface IEnumerable<out T> : IEnumerable  
{  
    IEnumerator<T> GetEnumerator();  
}  
  
public interface IEnumerator<out T> : IEnumerator  
{  
    bool MoveNext();  
    T Current { get; }  
}
```

Vous remarquez l'utilisation du mot clé *out* avant le type générique, qui permet de signifier que le type *T*, pourra uniquement être utilisé comme type de retour des méthodes définies dans ces interfaces. On dit alors que cette interface est « covariante » du type *T*. Ainsi toute énumération d'objets de type *A* (*IEnumerable<A>*) pourra être considérée comme une énumération d'objets de type *B* (*IEnumerable*), à condition que tout objet *A* puisse être converti en objet de type *B*.

Appliquons maintenant ce principe sur un cas pratique : voici un bloc de code permettant de créer une liste d'employés :

```
// C#  
  
// Variables locales;  
IList<Employe> oListeEmployes;  
  
// Création et alimentation de la liste des employes.  
oListeEmployes = new List<Employe>()  
{  
    new Employe("DURAND", "Alain", 2321.81),  
    new Employe("VIRON", "Karl", 1398.22),  
    new Employe("HOIN", "Pierre", 1210.09),  
    new Employe("HILL", "Tony", 3211.45),  
    new Employe("FROT", "Elise", 3232.90),  
    new Employe("ZERA", "Laurence", 2129.98)  
};
```

Maintenant, en C# 3.0, si nous écrivons ces instructions :

```
// C#  
  
IEnumerable<Personne> oListePersonnes;  
oListePersonnes = oListeEmployes.ToList<Personne>();
```

Nous obtenons le message d'erreur suivant « Argument d'instance : conversion impossible de 'System.Collections.Generic.IList<Test.Employe>' en 'System.Collections.Generic.IEnumerable<Test.Personne>' ».

Dans le langage C# 4.0, la co-variance permet de « convertir » une liste d'objets de type T en une énumération d'objets de type $T1$, si et seulement si une conversion est possible. En appliquant ce principe à notre exemple, la classe *Employe* dérivant de la classe *Personne*, il est maintenant possible de « convertir » une liste d'employés en une énumération de personnes. Ce constat permet alors d'écrire et d'exécuter en C# 4.0 les instructions suivantes :

```
// C#  
  
List<Personne> oListePersonnes;  
oListePersonnes = oListeEmployes.ToList<Personne>();
```

Plus concrètement, cette nouveauté se révèle pratique dans le cas suivant. Voici un bloc d'instructions permettant de créer deux listes d'objets distinctes : l'une de personnes et l'autre d'employés. Puis de fusionner ces deux listes pour n'en former qu'une seule, et d'afficher le nombre d'objets qu'elle contient :

```
// C#  
  
IList<Personne> oListePersonnes = new List<Personne>()  
{  
    new Personne("RAVAILLE", "James"),  
    new Personne("DOLLON", "Julien"),  
    new Personne("VERGNAULT", "Laurent")  
};  
  
IList<Employe> oListeEmployes = new List<Employe>()  
{  
    new Employe("ASSIS-RANTES", "Laurent", 2239.34),  
    new Employe("DORDOLO", "Matthieu", 1989.99)  
};  
  
var oListeFusion = oListePersonnes.Union(oListeEmployes);  
MessageBox.Show("Nombre d'éléments : " +  
oListeFusion.Count().ToString());
```

La co-variance est utilisée dans l'instruction `var oListeFusion = oListePersonnes.Union(oListeEmployes);` en permettant de convertir la liste `oListeEmployes` en `IEnumerable<Personne>`. Voici le résultat obtenu :



4.1.2 La contre-variance et les interfaces génériques

Dans le Framework .NET 4.0, l'interface générique *IEqualityComparer<T>* est définie de la manière suivante :

```
// C#  
  
public interface IEqualityComparer<in T>  
{  
    bool Equals(T x, T y);  
    int GetHashCode(T obj);  
}
```

Vous remarquez l'utilisation du mot clé *in* avant le type générique, qui permet de signifier que le type *T*, pourra uniquement être utilisé comme type des paramètres des méthodes de cette interface.

Grâce à cette écriture, un objet créé à partir d'une classe implémentant l'interface *IEqualityComparer<Object>*, pourra être considéré comme un objet de type *IEqualityComparer<string>*.

Définissons alors une classe nommée *PersonneComparer*, implémentant l'interface *IEqualityComparer<Personne>*. Cette classe permet de déterminer si deux instances de la classe *Personne* désigne la même personne :



```
// C#

public class PersonneComparer : IEqualityComparer<Personne>
{
    public bool Equals(Personne aPersonne1, Personne aPersonne2)
    {
        // Variables locales.
        bool bResult;

        // Initialisation.
        bResult = true;

        // Comparaison des références des deux personnes.
        if (aPersonne1 != aPersonne2)
        {
            // Cas des objets NULL.
            if (aPersonne1 == null || aPersonne2 == null)
                bResult = false;
            else
                // Les deux objets représentent la même personne s'ils ont le
                // même nom et même prénom.
                bResult = aPersonne1.Prenom == aPersonne2.Prenom &&
                aPersonne1.Nom == aPersonne2.Nom;
        }

        return bResult;
    }
}
```

```
// C#

public int GetHashCode(Personne aPersonne)
{
    // Variables locales.
    int iResult;

    // Initialisation.
    iResult = 0;

    // On vérifie que l'objet est différent de null.
    if (aPersonne != null)
    {
        // On obtient le HashCode du prénom s'il est différent de null.
        int iHashCodePrenom = 0;
        if (aPersonne.Prenom != null)
            iHashCodePrenom = aPersonne.Prenom.GetHashCode();

        // Calcul du hashCode du nom;
        int iHashCodeNom = 0;
        if (aPersonne.Nom != null)
            iHashCodeNom = aPersonne.Nom.GetHashCode();

        // Calcul du hascode de la personne à partir du nom et du prénom.
        iResult = iHashCodePrenom ^ iHashCodeNom;
    }

    return iResult;
}
}
```

Toute classe implémentant une interface doit proposer une implémentation des membres définis dans cette interface. En implémentant l'interface générique *IEqualityComparer<Personne>*, nous devons donc implémenter les membres :

- *Equals* : retourne une valeur booléenne indiquant si deux objets de type *Personne* représente la même personne. Ainsi, elle retourne *True* si les deux personnes passées en paramètre pointent vers la même instance ou s'ils ont le même nom et le même prénom.
- *GetHashCode* : permet d'obtenir le *HashCode* de l'objet de type *Personne* passé en paramètre, à partir de son nom et son prénom.

Voici un bloc d'instructions permettant de créer une liste d'employés, puis de l'apurer afin d'enlever les doublons, et d'afficher le nombre d'employés obtenus :

```
// C#  
  
private void CmdContreVariance_Click(object sender, EventArgs e)  
{  
    // Création de la liste des employés.  
    List<Employe> oListeEmployes = new List<Employe>()  
    {  
        new Employe("ASSIS-ARANTES", "Laurent", 2239.34),  
        new Employe("ASSIS-ARANTES", "Laurent", 2306.01),  
        new Employe("DORDOLO", "Matthieu", 1989.99)  
    };  
  
    // Suppression des doublons.  
    IEnumerable<Employe> oListeEmpl =  
        oListeEmployes.Distinct(new PersonneComparer());  
  
    // Affichage du nombre d'employés distincts.  
    MessageBox.Show(oListeEmpl.Count().ToString() + " employés.");  
}
```

Pour appliquer le concept de la contre-variance présenté ci-dessus, nous avons utilisé une instance de la classe *PersonneComparer*, permettant de comparer deux personnes, afin de comparer deux employés. Cette contre-variance est applicable car la classe *Employe* dérive de la classe *Personne*.

L'exécution du code ci-dessus affiche la boîte de message suivante :



5 Conclusion

Ce cours vous a présenté les principales nouveautés du langage C# 4.0 :

- Les paramètres nommés et les paramètres optionnels.
- Le typage dynamique.
- La co-variance et la contre-variance sur les classes génériques.

Ces nouveautés vous offriront plus de possibilités dans le développement de vos applications .NET avec le langage C#. Attention toutefois au typage dynamique : il est important de ne pas en abuser, au risque de créer des applications dans lesquelles des exceptions sont souvent levées...