



**Formation C# – Visual Studio - Delphi.NET/Win32  
Audit, Conseil, Développement**

© Copyright 2007 Olivier DAHAN  
Reproduction, utilisation et diffusion interdites sans l'autorisation de l'auteur. Pour plus  
d'information contacter odahan@e-naxos.com

## Les nouveautés de C# 3.0

---

C# est un langage en perpétuel mouvement. Changeant sans rien remettre en cause, s'améliorant au-delà de ce que bon nombre de développeurs pouvait même imaginer. Cette métamorphose pousse C# vers un langage fonctionnel supportant un style de plus en plus déclaratif. Bien entendu les avancées du langage font intégralement partie du bouillonnement général d'idées qui est celui des équipes Microsoft depuis ce que j'appellerais « l'ère .NET ».

En effet, depuis le lancement de la plateforme .NET (le Framework et C#), ce sont régulièrement des idées plus innovantes les unes que les autres que nous propose Microsoft. Qu'il s'agisse de WPF, WCF, WF, de Silverlight, ou bien de l'Entity Framework et donc aussi de LINQ (et je raccourci volontairement la liste à laquelle on pourrait ajouter Microsoft Ajax, Astoria, ...), chacune de ces évolutions pourrait à elle seule passer pour une révolution géniale chez un autre éditeur... Ne nous lassons pas par habitude ou tellement le rythme des nouveautés est soutenu et gardons intact notre capacité d'émerveillement ! Le Framework 3.5 regorge d'idées nouvelles, C# 3.0 n'est finalement qu'une partie de cette immense galaxie.

Je reparlerai d'ailleurs de l'Entity Framework dans un autre article et bien sûr de LINQ (même si ce dernier n'est pas uniquement lié aux données de type SQL), mais avant d'aborder ces avancées il est essentiel de bien comprendre les nouveautés de C# 3.0 tellement les évolutions du Framework sont intimement liées à celles du langage.

## Inférence des types locaux

Sous ce terme un peu barbare (*local type inference* en anglais) se cache une fonction puissante, celle du mot clé **var**.

**var** existe sous d'autres langages (JavaScript, Delphi) mais avec un comportement bien différent. Sous Delphi il s'agit du seul moyen de déclarer une variable, entre l'entête de méthode et son corps, sous JavaScript la variable déclarée par **var** n'est pas typée et peut contenir une chaîne de caractères comme un entier par exemple.

Sous C#, **var** s'utilise partout où une variable peut être déclarée (continuité du style) et s'emploie à la place du type car celui-ci sera deviné automatiquement en fonction de celui de l'expression (inférence du type). Une fois le type attribué il ne sera pas modifiable et la variable se comporte exactement comme si elle avait été déclarée de façon traditionnelle. Il ne faut d'ailleurs pas confondre ce fonctionnement avec les *variants* qu'on retrouve sous Delphi ou d'autres langages comme Visual Basic. Les variants possèdent un type dynamique fixé à l'exécution alors qu'une déclaration **var** de C# n'est qu'un raccourci qui donnera naissance à la compilation à une déclaration de type tout à fait classique.

Un exemple nous fera comprendre l'intérêt et la syntaxe de ce nouveau mot clé...

### Utilité

D'abord parlons de l'utilité, on retient mieux ce dont on comprend le sens et les avantages. Dans les langages objet à typage fort toute variable doit se déclarer avec son type. Mais comme toute variable objet doit être instanciée avant d'être utilisée on est très souvent obligé d'appeler le constructeur à la suite de la déclaration de la variable. De fait, on obtient une écriture du type de ceci :

```
MonTypeObjet unObjet = new MonTypeObjet() ;
```

Écrire deux fois « **MonTypeObjet** » dans une ligne si petite de code semble à la longue très contraignant. Certes la notation est rigoureuse mais bien peu efficace.

Regardons la même déclaration utilisant **var** :

```
var unObjet = new MonTypeObjet() ;
```

Du point de vu fonctionnel la variable « **unObjet** » sera identique et bien entendu toujours fortement typée. Il s'agit donc d'un artifice de notation qui rend le code plus lisible.

D'ailleurs il suffit de regarder le code IL généré à la compilation, il ressemble exactement à une déclaration de variable « classique », c'est-à-dire que le code IL contient le type inféré comme si celui-ci avait été saisi directement dans le code source.

### Exemples

Une fois l'intérêt mis en évidence, voici d'autres exemples d'utilisation :

```
var a = 3 ; // a sera de type int
var b = "Salut !" ; // b sera de type string
var q = 52.8 ; // q sera un double
```

```
var z = q / a ;           // z aussi
var m = b.Length() ;     // m sera un int
decimal d ; var f = d ;  // f sera un decimal
var o = default(string) ; // o sera un string
var t = null ;           // Interdit !
                           // le type ne peut pas être inféré...
```

On notera le mot clé `default(<type>)` qui retourne la valeur nulle par défaut pour le type considéré.

## Une porte sur les types anonymes

Arrivé à ce stade on pourrait se dire que, finalement, `var` n'a été introduit que pour pallier la paresse des développeurs... Même si cela n'est pas faux (mais un bon développeur doit être paresseux !), ce n'est pas que cela, `var` est aussi le seul moyen de déclarer des variables avec un **type anonyme**, autre nouveauté que nous verrons plus loin.

A noter, `var` ne peut être utilisé qu'à l'intérieur d'une portée locale. Cela signifie qu'on peut déclarer une variable locale avec `var` mais pas un paramètre de méthode ou un type de retour de méthode par exemple.

## Les expressions Lambda

Encore une terminologie savante qui peut effrayer certaines personnes. En réalité, et nous allons le voir, il n'y a rien d'inquiétant dans cette nouvelle syntaxe qui ne fait que poursuivre le chemin tracé par C# 2.0 et ses méthodes anonymes en simplifiant et en généralisant à l'extrême l'utilisation et la notation de ces dernières.

## Rappel sur les méthodes anonymes

Pour rappel, une méthode anonyme n'est qu'une méthode... sans nom, c'est-à-dire un morceau de code représentant le corps d'une méthode qu'on peut placer là où un pointeur de code (un *delegate*) est attendu, et ce, dans le respect de la déclaration du delegate (C# reste un langage très fortement typé malgré les apparences syntaxiques trompeuses de certaines de ses évolutions !).

Tout d'abord voyons un exemple de code utilisant une méthode anonyme à la façon de C# 2.0 :

```
public class DemoDelegate
{
    delegate T Func<T>(T a, T b);

    static T Agreger<T>(List<T> l, Func<T> f)
    {
        T result = default(T);
        bool premierPassage = true;
        foreach (T value in l)
        {
            if (premierPassage)
            {
                result = value;
            }
        }
    }
}
```

```

        premierPassage = false;
    }
    else
    {
        result = f(result, value);
    }
}
return result;
}

public static void LancerDemo()
{
    int somme;
    List<int> lesEntiers = new List<int> {1,2,3,4,5,6,7,8,9};
    somme = DemoDelegate.Agreger(
        lesEntiers,
        delegate(int a, int b) { return a + b; }
    );
    Console.WriteLine("La somme = {0}", somme);
}

static void Main(string[] args)
{
    DemoDelegate.LancerDemo();
}

```

Dans cet exemple nous définissons une classe **DemoDelegate** qui expose une méthode générique **Agreger** permettant d'appliquer une fonction sur une liste dont les éléments peuvent être de tout type. Liste et fonction à appliquer étant passées en paramètre lors de l'appel de la méthode.

La méthode (statique) **LancerDemo** crée une liste d'entiers ainsi qu'une variable **somme**. Ensuite elle assigne à cette dernière le résultat de l'appel à **DemoDelegate.Agreger** en lui passant en paramètre la liste d'entiers ainsi qu'une méthode anonyme (en surligné gris dans l'exemple ci-dessus) définissant le traitement à effectuer.

On notera que cette méthode anonyme répond au prototype du delegate **Func** aussi déclaré dans **DemoDelegate**. Cette déclaration permet d'assurer un typage fort du code passé. En place et lieu d'une méthode anonyme de C# 2.0 nous aurions été obligés, sous C# 1.0, de créer une méthode de calcul portant un nom et de passer ce dernier en argument de la méthode **Agreger** via un delegate.

On remarque ainsi que le passage de C# 1.0 à C# 2.0 nous a permis une économie syntaxique rendant le code plus léger, plus élégant et plus facilement réutilisable, et ce, grâce aux méthodes anonymes et aux génériques.

Peut-on aller plus loin dans le même esprit ?

### Aller plus loin grâce aux expressions Lambda

La réponse de C# 3.0 est claire : oui, et cela s'appelle les expressions Lambda.

Regardons comment à l'aide des expressions Lambda nous pouvons réécrire le code de la méthode **LancerDemo** :

```
public static void LancerDemoCS3()
{
    int somme;
    List<int> lesEntiers = new List<int> {1,2,3,4,5,6,7,8,9};
    somme = DemoDelegate.Agreger(lesEntiers,
        (int x, int y) => { return x + y; });
    Console.WriteLine("La somme = {0}", somme);
}
```

Le code surligné en gris contient l'expression Lambda.

Comme on le voit il n'y a rien de bien compliqué, nous n'avons fait que supprimer le mot clé **delegate** et avons introduit le symbole **=>** entre la déclaration des paramètres de la méthode anonyme et l'écriture de son code.

On peut lire l'expression ci-dessus de la façon suivante : « *étant donné les paramètres x et y de type entier, retourner la somme de x et y.* »

La simplification peut aller plus loin car les expressions lambda supportent le *typage implicite* des paramètres. Ainsi l'expression peut être simplifiée comme suit :

```
(x, y) => { return x + y; }
```

Le type des paramètres **x** et **y** peut être omis puisque C# sait que les paramètres doivent correspondre au prototype **Func**. Certes ce delegate est totalement déclaré avec des types génériques... Et c'est en réalité par l'appel de **Agreger** et grâce au premier paramètre (la liste de valeurs) que C# 3.0 peut inférer les types. Il n'y a donc aucune magie et surtout, tout reste très fortement typé sans faire aucune concession.

L'inférence des types est effectuée à la compilation, bien entendu, et non à l'exécution.

Comment prononcer le nouveau symbole ?

Il n'y a semble-t-il pas de nom particulier pour le signe **=>** des expressions Lambda. On peut proposer de le lire comme « Tel Que » lorsque l'expression est un prédicat ou « Deviens » lorsqu'il s'agit d'une projection.

*Pour rappel, un prédicat est une expression booléenne généralement utilisée pour créer un filtre et une projection est une expression retournant un type différent de son unique paramètre.*

## Deux écritures possibles du corps

La syntaxe d'une expression Lambda supporte deux façons de définir le corps de la méthode anonyme, soit avec des *brackets* comme nous l'avons vu dans l'exemple ci-dessus, soit sous la forme d'une simple instruction **return** en omettant ce mot-clé. Ainsi l'expression de notre exemple deviendra encore plus simplement :

```
somme = DemoDelegate.Agreger(lesEntiers, (x, y) => x + y );
```

Nous avons supprimé les brackets et même le point-virgule final puisque nous ne sommes plus dans un bloc de code mais plutôt dans l'écriture d'une simple instruction et que la syntaxe à cet endroit n'autorise pas de point-virgule après l'instruction (l'expression Lambda occupe en effet la place du second paramètre de **Agreger**, les paramètres sont seulement suivis par des virgules sauf le dernier, ce qui est le cas de l'expression ici).

## Simplifions encore

Prenons maintenant un autre exemple réclamant un delegate ne possédant qu'un seul paramètre :

```
public delegate T Func2<T>(T x);

public static T Agreger2<T>(List<T> l, Func2<T> f)
{
    T result = default(T);
    foreach (T value in l) result = f(value);
    return result;
}

public static void LancerDemoCS3v4()
{
    Single somme=0f;
    List<Single> lesSimples =
        new List<Single> { 1.5f, 2.6f, 3.7f, 4.8f, 5.9f,
            6.0f, 7.1f, 8.2f, 9.3f };
    somme = DemoDelegate.Agreger2(lesSimples, (x) => somme += x);
    Console.WriteLine("La somme = {0}", somme);
}
```

Dans la version ci-dessus nous avons créé un nouveau delegate qui ne prend qu'un seul paramètre (**Func2**). La méthode **Agreger2** a été modifiée pour refléter cette modification, son code est devenu d'ailleurs plus simple.

Mais ce qui nous intéresse est l'utilisation de la méthode Lambda (sur fond gris).

En dehors du fait qu'elle n'utilise plus qu'un seul paramètre, conformément au nouveau delegate, on s'aperçoit qu'elle peut se permettre d'utiliser la variable locale **somme** à l'intérieur même de sa définition. En effet, **somme** est une locale de la méthode contenant l'expression et sa portée ainsi que sa durée de vie sont étendues à l'instance de la méthode anonyme définit par l'expression Lambda.

Simplifions encore... Lorsqu'il n'y a qu'un seul paramètre comme dans le dernier exemple on peut omettre les parenthèses qui l'entourent. Ainsi, l'expression pourra directement s'écrire :

```
(x) => somme += x
```

## Rappel important

Les expressions lambda, tout comme les méthodes anonymes dont elles sont un prolongement et une simplification syntaxique, ne servent à saisir que des petits bouts de code et non des pages entières ! On les utilise principalement pour créer des filtres ou autres fonctions de ce type ne réclamant que quelques instructions au maximum. Si le corps d'une

expression Lambda, tout comme une méthode anonyme C# 2.0, dépasse cette limite il faut alors déclarer une méthode et utiliser un delegate « classique ». Que ceux qui seraient tentés d'écrire trois pages de code dans une expression Lambda soient prévenus, cela est très fortement déconseillé !

## Un autre exemple

Cette mise au point indispensable effectuée, voyons comment une expression Lambda peut simplifier grandement un code de type filtrage de liste (donc utiliser l'expression en tant que prédicat).

```
public class DemoPredicat
{
    public static void AfficheListe<T>(T[] items, Func<T, bool> leFiltre)
    {
        foreach (T item in items) if (leFiltre(item)) Console.WriteLine(item);
    }

    public static void lancerDemo()
    {
        string[] villes = { "Paris", "Berlin", "Londres", "New-york",
                            "Barcelone", "Milan" };

        Console.WriteLine("Les villes sans 'e' dans leur nom sont:");
        AfficheListe(villes, s => !s.Contains('e'));

        Console.WriteLine("Les villes ayant 'i' dans leur nom sont:");
        AfficheListe(villes, s => s.Contains('i'));
    }
}
```

Dans le code ci-dessus que remarque-t-on ?

D'abord nous n'avons pas déclaré de delegate. Nous avons utilisé une déclaration existante dans le Framework. Ce genre de prototype étant très courant, notamment pour les prédicats, le Framework le contient déjà.

Ensuite nous voyons très clairement à quel point les expressions Lambda rendent le code concis et clair. La liste des villes est filtrée et affichée deux fois, les villes ne possédant pas de « e » dans leur nom puis celles contenant un « i » dans ce même nom. D'ailleurs la même fonction **AfficherListe** pourrait être utilisée sans modification pour afficher une liste d'entiers ou de dates filtrés puisqu'elle n'utilise que des types génériques. Quant à l'appel de cette méthode, il contient directement le filtrage à effectuer, de façon simple, clair et lisible, sans artifice ni delegate superflu !

Les expressions Lambda, c'est exactement ça : plus de simplicité et d'élégance pour un code plus lisible, plus flexible et plus puissant.

On notera que le Framework définit l'ensemble suivant de delegates utilisables de la même façon que **Func** dans notre exemple qu'on retrouve en première entrée de la liste :

- public delegate T Func< T >();
- public delegate T Func< A0, T >( A0 arg0 );

- `public delegate T Func<A0, A1, T> ( A0 arg0, A1 arg1 );`
- `public delegate T Func<A0, A1, A2, T >( A0 arg0, A1 arg1, A2 arg2 );`
- `public delegate T Func<A0, A1, A3, T> ( A0 arg0, A1 arg1, A2 arg2, A3 arg3 );`

Il n'y a bien entendu aucune obligation d'utiliser ces types définis dans **System.Linq** (ajouté automatiquement aux projets sous VS 2008). Vous pouvez utiliser vos propres types. Il n'y a qu'un seul cas dans lequel il faut respecter les définitions de delegate présentées ci-dessus : lorsqu'on veut transformer une expression en **arbre d'expression**.

## Les arbres d'expression

Il faut bien prendre conscience que ces ajouts au langage ont été faits certes pour leur puissance intrinsèque mais aussi et surtout pour faciliter l'implémentation de Linq... Or Linq, dont nous parlerons en détail dans un prochain article, impose certaines exigences comme le fait de pouvoir transformer une expression en un arbre facilement navigable. Les arbres d'expression sont analysés à l'exécution et peuvent même être créés à ce moment. Cela est utilisable de plusieurs façons, Linq, lui, s'en sert notamment pour transformer les requêtes Linq C# en syntaxe SQL (Linq to ADO.NET) conforme à la base cible. Cette dernière dépendant de la connexion et de la base cible, de son langage, des champs, Linq a besoin d'interpréter les arbres à ce moment et non à la compilation.

La différence principale entre une expression Lambda comme celles que nous avons vues dans cet article et un arbre expression se situe uniquement dans la représentation de la méthode anonyme. Une expression Lambda binaire est compilée et se présente sous la forme de code IL, alors qu'un arbre expression est une représentation mémoire dynamique (modifiable notamment) donc une représentation runtime.

Seules les expressions Lambda possédant un corps peuvent être transformées en arbre expression. Nous avons vu dans cet article que dans des cas très simples les expressions pouvaient s'écrire comme une instruction, ce sont les expressions sous cette forme qui sont exclues de la transformation en arbre. Nous aborderons les arbres d'expression dans un prochain article, le sujet réclamant de s'y attarder plus longuement.

## Les méthodes d'extension

On peut les voir comme une émanation de la design pattern *Decorator*. On les trouvait déjà sous d'autres formes dans d'autres langages, par exemple Borland les a implémentées dans Delphi 8 pour .NET principalement pour ajouter artificiellement les méthodes de TObject à System.Object de .NET et faire passer le second pour le premier, fonctionnellement, aux yeux de la VCL.NET.

Il s'agit donc de pouvoir ajouter des méthodes à une classe sans modifier la dite classe... Magique ? Oui et non. Cela peut paraître très rusé mais risque vite d'être ingérable si on imagine un code utilisant en plus des interfaces et de l'héritage cette technique qui fait sortir des méthodes du chapeau du magicien et non des classes elles-mêmes... Vous voilà prévenus, cela peut être utile, mais c'est à utiliser avec une grande modération !

Microsoft a implémenté cette possibilité dans C# 3.0 pour simplifier la syntaxe de Linq, la rendre plus lisible et plus concise.

C# 3.0 fait en sorte que les méthodes accrochées à une classe ne puissent accéder qu'à ces membres publics. De fait le procédé ne permet en aucune sorte de violer le principe d'encapsulation des objets. Cela a l'avantage d'être propre et d'éviter certaines dérives.

Les class helpers doivent être définis dans des classes statiques avec des méthodes statiques.

Je n'ai hélas trouvé aucune utilisation simple et pertinente des class helpers, rien qui ne puisse être réglé bien plus proprement par l'héritage ou le support d'une interface. Vous l'avez compris, je n'aime pas trop cette « amélioration » du langage. Mais personne ne m'oblige à m'en servir non plus, alors tout va bien !

C# étant un langage très puissant il ne faut pas non plus regarder sa syntaxe par le très réducteur gros bout de la lorgnette... Comme simple possibilité, les class helpers ne sont pas indispensables, toutefois lorsqu'on associe class helpers et généricité, on peut arriver à trouver des utilisations intelligentes et élégantes, c'est d'ailleurs dans un tel esprit que Linq s'en sert. A vous de trouver des utilisations au moins aussi pertinentes.

Sans trop entrer dans de tels détails (je reste pour l'instant, et faute de recul, réservé sur le sujet des class helpers au sein d'un code bien écrit et maintenable) je vous livre un exemple pour que vous puissiez en comprendre le mécanisme :

```
public struct Article
{
    public int Code;
    public string Désignation;
    public override string ToString()
    { return Code + ", " + Désignation; }
    public Article(int code, string designation)
    { Code = code; Désignation = designation; }
}

public struct Client
{
    public int Code;
    public string Société;
    public override string ToString()
    { return Code + ", " + Société; }
    public Client(int code, string société)
    { Code = code; Société = société; }
}

public static class DemoHelpers
{
    public static void LanceDemo()
    {
        Article a = new Article(101, "Zune 20 Go");
        Client c = new Client(5800, "E-Naxos");
        a.Affiche(); // appel « magique » à Affiche
        c.Affiche();
    }
}
```

```
// déclaré non imbriqué dans une autre classe
public static class Afficheur
{
    public static void Affiche(this object o)
    { Console.WriteLine(o.ToString()); }
}
```

Dans l'exemple ci-dessus deux structures sont déclarées, **Article** et **Client**, chacune ayant ses spécificités et ne partageant rien en commun. Ailleurs dans le code est déclarée la classe statique **Afficheur** qui possède la méthode **Affiche** (statique aussi). Les paramètres de **Affiche**, et l'utilisation de **this**, en font automatiquement un class helper.

C'est ce qui permet d'appeler **Affiche** depuis des instances de **Article** ou de **Client**. Si la déclaration de la méthode **Affiche** avait indiqué **Article** à la place de **object**, seule la classe **Article** aurait pu utiliser **Affiche** qui ne serait donc plus visible depuis les instances de **Client**.

## Les expressions d'initialisation des objets

Un code source travaille sur des variables qu'il faut déclarer et initialiser. La syntaxe dédiée à ces opérations élémentaires est importante puisque, revenant très souvent sous les doigts du développeur, toute lourdeur sera ressentie comme pénible avec le temps.

## Les initialisations rapides de C# 1.x

C# 1.0 proposait déjà quelques facilités syntaxiques, pour rappel :

```
string s = "Bonjour" ;
single x = 10.0f ;
Synthé synthé = new Synthé("Prophet",5,"Sequential Circuit") ;
```

Lorsqu'on instancie un type par valeur ou par référence on peut appeler l'un de ses constructeurs permettant, en une seule opération, d'initialiser les principaux états de l'objet. C'est le cas du dernier exemple ci-dessus.

Si cette approche est particulièrement efficace elle oblige à prévoir (et à coder) de nombreuses surcharges du constructeur dans chaque classe. On remarque avec *Intellisense* que Microsoft a utilisé cette façon de faire en de nombreuses occasions ce qui facilite grandement le codage. Certaines classes possèdent jusqu'à dix ou vingt constructeurs différents... Autant de code à écrire et à maintenir malgré tout.

## Les initialisations avec la syntaxe de base

Si on en revient à la syntaxe de base pour créer et initialiser un objet et que nous reprenons notre dernier exemple, le code s'écrirait comme suit :

```
Synthé synthé ;
synthé = new Synthé() ;
synthé.Modèle= "Prophet" ;
synthé.Version = 5 ;
synthé.Fabriquant = "Sequential Circuit" ;
```

On notera la lourdeur du style, et l'augmentation du nombre de bogues potentiels comparativement à la syntaxe raccourcie vue plus haut.

### C# 3.0, le meilleur des deux mondes

C# 3.0 apporte le meilleur des deux syntaxes, celle de l'appel à un initialiseur, compacte, et celle plus classique de l'accès à chaque membre, plus complète et ne réclamant pas l'écriture d'une série d'initialiseurs spécialisés pour chaque cas de figure.

Reprenons l'exemple de notre bon vieux synthétiseur...

```
Synthé synthé = new Synthé
  {Modèle="Prophet",Version=5,Fabriquant="Sequential Circuit",
   AnnéeDeSortie = 1978 } ;
```

On remarque immédiatement les avantages, par exemple nous avons pu initialiser l'année de sortie alors qu'elle n'a pas été prévue dans le constructeur / initialiseur de cette classe. De fait aucun initialiseur n'a été codé dans la classe d'ailleurs. On remarque ensuite qu'on appelle le constructeur par défaut en omettant les parenthèses, il est directement suivi du bloc d'initialisation entre brackets.

La technique est séduisante et permet de gagner en concision, toutefois elle n'est pas parfaite. Il ne faut donc pas voir cette solution comme la fin des initialiseurs spécifiques mais plutôt comme un complément pratique, parfois tout à fait suffisant, parfois ne pouvant répondre à tous les besoins d'un vrai initialiseur.

Par exemple seules les propriétés publiques sont accessibles, il est donc impossible par cette syntaxe d'initialiser un état interne à partir des paramètres d'initialisation, ce qu'un constructeur permet de faire. Enfin, puisqu'on accède aux propriétés publiques et que très souvent les modifications de celles-ci déclenchent des comportements spécifiques (mise à jour de l'affichage par exemple), la nouvelle syntaxe peut s'avérer pénalisante là où un constructeur est capable de changer tous les états internes avant d'accomplir les actions ad hoc.

Il faut comprendre en effet que la nouvelle syntaxe des initialiseurs d'objets n'est qu'un artifice syntaxique, il n'y a pas eu de changement du langage lui-même et le code IL produit par la nouvelle syntaxe est rigoureusement identique à la syntaxe de base pour créer et initialiser un objet (déclaration de la variable, appel du constructeur puis initialisation de chaque propriété, voir l'exemple plus haut).

### L'appel aux constructeurs

La nouvelle syntaxe est en revanche assez souple pour permettre d'appeler un autre constructeur que celui par défaut, et dans un tel cas elle procure bien un avantage stylistique. Toujours en partant du même exemple :

```
Synthé synthé = new Synthé("Prophet",5,"Sequential Circuit")
  { AnnéeDeSortie = 1978 } ;
```

Ici nous supposons qu'il existe un constructeur prenant en compte le modèle, la version et le fabricant. Toutefois il n'en existe aucune version permettant aussi d'initialiser l'année de sortie, comme cette propriété publique existe il est possible de mixer l'appel au constructeur le plus proche de notre besoin et d'ajouter à la suite l'initialisation du ou des champs qui nous intéressent (**AnnéeDeSortie** ici).

Les expressions d'initialisation des objets ne sont pas un ajout décisif au langage mais habilement utilisées elles complètent la syntaxe existante pour produire un code toujours plus clair, lisible et plus facilement maintenable. On notera que C# fait toujours les choses avec beaucoup de précautions puisque l'appel à une expression d'initialisation crée une variable cachée en mémoire jusqu'à ce que toutes les initialisations soient terminées et uniquement à ce moment là la variable est renseignée (la variable **synthé** dans notre exemple). Ainsi on retrouve les avantages d'un constructeur, tout est passé ou rien n'est passé, mais à aucun moment un morceau de code ne pourra accéder à une variable « à demi » initialisée.

### Un peu de magie...

Nous avons défini la classe **Synthé** pour l'exemple plus haut, nous allons la réutiliser pour créer une classe **MiniStudio** qui définit un synthétiseur principal et un autre, secondaire :

```
public class MiniStudio
{
    private Synthé synthéPrincipal = new Synthé();
    private Synthé synthéSecondaire = new Synthé();
    public Synthé SynthéPrincipal { get { return synthéPrincipal; } }
    public Synthé SynthéSecondaire { get { return synthéSecondaire; } }
}
```

Comme on le voit ci-dessus, les deux synthés sont définis comme des champs privés de la classe auxquels on accède via des propriétés en lecture seule.

Nous pouvons dès lors instancier et initialiser un **MiniStudio** de la façon suivante en utilisant la nouvelle syntaxe :

```
var studio = new MiniStudio
{
    SynthéPrincipal = { Modèle = "Prophet", Version = 5 },
    SynthéSecondaire = { Modèle = "Wave", Version = 2,
                        Fabriquant="PPG", AnnéeDeSortie = 1981 }
};
```

Il est donc tout à fait possible d'une part d'utiliser la nouvelle syntaxe pour initialiser des instances imbriquées (les instances de **Synthé** dans l'instance de **MiniStudio**), mais surtout on peut voir que les propriétés **SynthéPrincipal** et **SynthéSecondaire** sont accessibles en écriture alors qu'elles sont en lecture seule ! Violation de l'encapsulation, magie noire ?

Bien sur que non. Et heureusement... En réalité nous n'écrivons pas une nouvelle valeur pour les pointeurs d'objet que sont les propriétés **SynthéPrincipal** et

**SynthéSecondaire**, nous ne faisons qu'accéder aux instances créées par la classe **MiniStudio** et aux propriétés publiques de ces instances...

Toutefois, si nous avons fait précéder les brackets par **new** le compilateur aurait rejeté la syntaxe puisque ici il n'est pas possible de passer une nouvelle instance aux synthés (les propriétés sont bien en lecture seule).

De fait la ligne suivante serait rejetée à la compilation :

```
var studio = new MiniStudio
    {
        SynthéPrincipal = new { Modèle = "Prophet", Version = 5 },
        SynthéSecondaire = { Modèle = "Wave", Version = 2,
                            Fabriquant="PPG", AnnéeDeSortie = 1981 }
    };
```

L'erreur rapportée est "Error 1, Property or indexer 'SynthéPrincipal' cannot be assigned to -- it is read only"

La nouvelle syntaxe est utilisable aussi pour initialiser des collections tant qu'elle supporte l'interface **System.Collection.Generic, ICollection<T>**. Dans ce cas les items seront initialisés et **ICollection<T>.Add(T)** sera appelé automatiquement pour les ajouter à la liste.

Il est ainsi possible d'avoir des initialisations imbriquées (comme l'exemple du mini studio) au sein d'initialisation de collections et inversement ainsi que toute combinaison qu'on peut imaginer. Bien plus qu'un simple artifice, on se rend compte que les expressions d'initialisation d'objets ne sont en réalité pas si anecdotique que ça, même si comparée aux autres nouveautés de C# 3.0 elles semblent moins essentielles.

Produire un code clair, lisible et maintenable est certes moins éblouissant de prime abord que de montrer des expressions Lambda, mais au bout du compte c'est peut-être ce qui est le plus important en production...

## Les types anonymes

Partons maintenant pour la cinquième dimension, *twilight zone* !, et abordons ce qui semble un contresens dans un langage très fortement typé : les types.. anonymes.

Imaginons une instance d'une classe qui n'a jamais été définie mais qui peut malgré tout posséder des propriétés (que personne n'a créées) auxquelles on peut accéder !

Regardons le code suivant :

```
var truc = new { Couleur = Color.Red, Forme = "Carré" };
var bidule = new { Marque = "Intel", Type = "Xeon", Coeurs = 4 };

Console.WriteLine("la couleur du truc est " + truc.Couleur +
                  " et sa forme est un " + truc.Forme);
Console.WriteLine("le processeur est un " + bidule.Type + " de chez " +
                  bidule.Marque + " avec " + bidule.Coeurs + " coeurs.");
```

Ce code produira la sortie suivante :

```
la couleur du truc est Color [Red] et sa forme est un Carré  
le processeur est un Xeon de chez Intel avec 4 coeurs.
```

C# créé bien des classes (un type) pour chacune des variables (**truc** et **bidule** dans l'exemple ci-dessus). Si on affiche le type (par **GetType()**) de ces dernières on obtient :

```
le type de truc est <>f__AnonymousType0`2[System.Drawing.Color,System.String]  
le type de bidule est  
<>f__AnonymousType1`3[System.String,System.String,System.Int32]
```

Plus fort, si nous créons un objet **trucbis** défini de la même façon que l'objet **truc** et que nous inspectons les types, nous trouverons le même type que l'objet **truc**, ce qui les rend compatibles pour d'éventuelles manipulations groupées ! Les propriétés doivent être définies dans le même ordre, si nous inversions **Couleur** et **Forme**, un nouveau type sera créé par le compilateur.

Normalement sous C# nous ne sommes pas habitués à ce que l'ordre des membres dans une classe ait une importance, mais il faut bien concevoir que tous les nouveaux éléments syntaxiques de C# 3.0 servent en réalité à l'implémentation de Linq, et Linq a besoin de pouvoir créer de types qui n'existent pas, par exemple créer un objet qui représente chaque ligne du résultat d'un SELECT, et il a besoin de faire la différence dans l'ordre des champs puisque dans un SELECT l'ordre peut avoir une importance.

## Conclusion

Les nouveaux éléments syntaxiques de C# 3.0 ne s'arrêtent pas là puisqu'il y a aussi tout ce qui concerne Linq et le requêtage des données. Mais avant d'aborder Linq dans un prochain article il était essentiel de fixer les choses sur les nouvelles syntaxes introduites justement pour servir Linq.

Une fois les types anonymes compris, les expressions Lambda digérées et tout le reste, il vous semblera plus facile d'aborder la syntaxe et surtout l'utilisation de Linq qui fait une utilisation débridée de ces nouveautés !

Espérant vous avoir éclairé utilement, je vous souhaite un happy coding !

Olivier Dahan

[www.e-naxos.com](http://www.e-naxos.com)

[odahan@e-naxos.com](mailto:odahan@e-naxos.com)

Code source : Le fichier ArticleNewCS30.ZIP accompagne cet article. Il contient un projet console qui démontre tous les aspects syntaxiques abordés ici.