

# Le mot clé yield et les itérateurs en C#

par Romain Verdier (<http://codingly.com>)

Date de publication : 28 mai 2008


Dernière mise à jour :

Cet article explore en profondeur les itérateurs apparus avec C# 2.0, ainsi que l'utilisation du mot clé yield.

---

1 - Introduction.....	3
2 - Rappel : le pattern Iterator selon .NET.....	3
3 - Dissection de yield.....	4
3.1 - Qu'est-ce qu'un itérateur ?.....	4
3.2 - Usage de yield.....	5
3.3 - Principe.....	5
3.4 - Enumerable et Enumerator object.....	6
4 - Quelques exemples.....	7
4-1 - Nombres pairs.....	7
4.2 - Yield c'est rigolo.....	7
4.3 - Yield break.....	8
4.4 - Valeurs calculées.....	8
5 - Exécution différée et conséquences.....	8
6 - Pour aller plus loin.....	10
7 - Quelques restrictions.....	13
8 - Conclusion.....	14

## 1 - Introduction

 *A l'origine, cet article a été rédigé pour mon **blog**. Je vous le livre ici sans l'avoir modifié, ce qui peut parfois expliquer certaines phrases étranges ou déplacées. Je vous prie de m'en excuser par avance, et espère que vous arriverez à en faire abstraction.*

Cet article est un peu particulier. D'une part, il s'agit de mon premier vrai post, et d'autre part, j'ai choisi de traiter en détail un sujet pas forcément nouveau et surtout très spécifique : les itérateurs et le mot clé yield de C#.

C'est pas ma faute, tout le monde (ou presque) s'en fout, ou ne sait pas qu'il existe.

Pourtant, ce mécanisme n'est pas seulement troublant, il est également puissant lorsqu'on l'utilise en maîtrisant son fonctionnement. S'il me fallait trouver un exemple, je parlerais de l'implémentation principale de LINQ, basée sur les itérateurs et le mot clé yield.

Tout ce qui va suivre n'est pas forcément utile si vous cherchez simplement à savoir comment utiliser le mot clé yield. Par contre, si vous cherchez à comprendre le fonctionnement réel des itérateurs, il peut s'agir d'un bon point de départ. C'est un peu le parallèle que l'on pourrait faire entre la MSDN et le livre des spécifications de C#.

## 2 - Rappel : le pattern Iterator selon .NET

En .NET, ce **pattern** est quasiment natif, puisque le framework propose deux interfaces incontournables : **IEnumerable** et **IEnumerator**. Depuis C# 2.0 on trouve également les versions génériques : **IEnumerable<T>** et **IEnumerator<T>** :

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}

public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
    void Reset();
}

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

public interface IEnumerator<T> : IEnumerator, IDisposable
{
    T Current { get; }
}
```

Quand je dis qu'elles sont incontournables, ce n'est pas exagéré. Toutes les collections du framework les implémentent (y compris les tableaux) et il existe même en C# un mot clé permettant d'itérer sur ces collections : **foreach**.

```
// ferns est ici une collection quelconque implémentant IEnumerable.
foreach(Fern fern in ferns)
{
    Console.WriteLine(fern);
}
```

On pense souvent que foreach ne peut être utilisé que sur des IEnumerable mais ce n'est pas tout à fait exact. Le mot clé fonctionne sur n'importe quel type possédant une méthode publique GetEnumerator retournant un IEnumerator, même si ce type n'est pas une implémentation d'IEnumerable.

Et que fait réellement le foreach ? Il suffit de demander à **Reflector**. En gros, l'itération utilisant le foreach précédent produit le code IL correspondant à :

```
IEnumerator enumerator = ferns.GetEnumerator();
try
{
    while (enumerator.MoveNext())
    {
        object current = enumerator.Current;
        Console.WriteLine(current);
    }
}
finally
{
    if (enumerator is IDisposable)
    {
        ((IDisposable) enumerator).Dispose();
    }
}
```

On note qu'il n'y a rien de magique, et que le code produit est identique à celui qu'on aurait pu écrire à l'aide d'un while. Il est intéressant de constater qu'il n'y a effectivement aucun lien entre le foreach et l'interface IEnumerable. Le compilateur C# se contente d'inspecter le type sur lequel itérer pour récupérer les métadonnées relatives à une méthode GetEnumerator. S'il ne les trouve pas, il lève une erreur durant la génération, et s'il les trouve, il produit simplement le bytecode d'appel à cette méthode. IEnumerable est donc dans le cas précis du foreach une abstraction impunément méprisée par le compilateur.

Lorsqu'on cherche à implémenter le pattern iterator en .NET, la solution la plus sage est donc de s'appuyer sur les interfaces du framework. En gros, pour rendre un type énumérable, on doit faire deux choses :

- Implémenter IEnumerable ou IEnumerable<T> au niveau du type à itérer.
- Implémenter un itérateur qui respecte l'interface IEnumerator ou IEnumerator<T>, respectivement.

Généralement, on ne se choisit pas d'implémenter uniquement ces interfaces, mais plutôt les interfaces des collections de base du framework, qui implémentent elles-mêmes IEnumerable et/ou IEnumerable<T> : ICollection, IList, IDictionary, etc. Je ne vais pas surcharger le post en publiant un exemple, mais si vous en voulez un significatif vous pouvez utiliser Reflector pour désassembler la classe générique List<T> du framework et son Enumerator<T> imbriqué. Vous verrez que le code nécessaire à la gestion des itérations est simple, mais nécessite tout de même une soixantaine de lignes.

### 3 - Dissection de yield

Dans le titre de l'article, #yield# est placé avant #itérateurs#. C'est une forme de marketing. En réalité, il est plutôt question ici des itérateurs (iterators), qui constituent une fonctionnalité de C# 2.0. Il est important de préciser à ce stade que lorsqu'on parle d'itérateurs, on ne parle pas des types implémentant IEnumerator, mais bien d'un mécanisme propre au langage C# basé sur l'utilisation d'un nouveau mot clé : yield.

#### 3.1 - Qu'est-ce qu'un itérateur ?

Un itérateur est une méthode, un opérateur ou un getter dont :

- Le type de retour est un des suivants : IEnumerable, IEnumerable<T>, IEnumerator, IEnumerator<T>.

- Le corps contient le mot clé yield.

Par convention, on considère que l'implémentation d'une telle méthode (ou opérateur, ou getter) est un bloc itérateur. Un **bloc itérateur** a pour but de produire une séquence de valeurs du même type. Le type des valeurs retournées est appelé **yield type**. Lorsque le bloc itérateur retourne un IEnumerable ou un IEnumerator, le *yield type* est object. Lorsque le bloc itérateur retourne un IEnumerable<T> ou un IEnumerator<T>, le *yield type* est T.

### 3.2 - Usage de yield

Un bloc itérateur est caractérisé par l'utilisation du mot clé yield. Ce dernier peut apparaître sous deux formes différentes. Je les cite maintenant, mais nous détaillerons plus tard :

- yield return <expression>;
- yield break;

Il faut juste garder en tête que dans la première forme, le *yield type* du bloc itérateur doit être assignable à partir du type de <expression>.

En introduction, et pour illustrer son utilisation, prenons un exemple simple : une méthode capable de retourner les 10 premiers entiers strictement positifs. L'implémentation la plus naïve qui me vient à l'esprit est la suivante :

```
public IEnumerable<int> GetFirstTenIntegers()
{
    List<int> result = new List<int>();
    for (int i = 1; i <= 10; i++)
    {
        result.Add(i);
    }
    return result;
}
```

Avec yield, il devient possible d'écrire :

```
public IEnumerable<int> GetFirstTenIntegers()
{
    for (int i = 1; i <= 10; i++)
    {
        yield return i;
    }
}
```

### 3.3 - Principe

Essayons à présent de comprendre comment fonctionnent les itérateurs en C# et que signifie le mot clé yield. Avant C# 2.0, cette notion n'existait pas donc le mot clé n'existait pas. Il s'agit maintenant d'un mot clé **contextuel**, pour des raisons de compatibilité descendante. Un mot clé contextuel est un mot clé qui revêt sa signification spéciale uniquement dans un contexte particulier. Ici, yield est interprété comme un mot clé lorsqu'il est utilisé avant return ou break, mais sinon il est tout à fait possible de l'utiliser comme identifiant pour une variable, un membre, un type, etc.

La chose la plus importante à savoir quant au fonctionnement des itérateurs est qu'un bloc itérateur n'est pas un bloc de code qui sera exécuté, mais un bloc de code qui sera **interprété par le compilateur pour générer du code**. Il ne s'agit pas d'une simple nuance, et je vais essayer de détailler.

Durant la génération, voici ce qui se produit lorsque le compilateur C# rencontre un bloc itérateur :

- 1 Si le type de retour du bloc est `IEnumerator` ou `IEnumerator<T>`, le compilateur fera en sorte qu'un **objet énumérateur** soit instancié et retourné. Cet objet implémentera respectivement `IEnumerator` ou `IEnumerator<T>`.
- 2 Si le type de retour du bloc est `IEnumerable` ou `IEnumerable<T>`, le compilateur fera en sorte qu'un **objet énumérable** soit instancié et retourné. Cet objet implémentera respectivement `IEnumerable` et `IEnumerator`, ou bien `IEnumerable<T>` et `IEnumerator<T>`, ce qui fait en fait également un objet énumérateur.

Dans tous les cas, le compilateur doit générer un type pour ces objets durant la compilation. Il est facile de deviner que tout le secret du mécanisme des itérateurs réside donc dans le processus de génération de ces types.

### 3.4 - Enumerable et Enumerator object

Résumons. Lorsque qu'un bloc itérateur est interprété, le compilateur doit générer un type d'objet énumérable ou un type d'objet énumérateur. Dans les deux cas, il doit générer l'implémentation d'`IEnumerator` et `IEnumerator<T>`, puisque un type d'objet énumérable est **aussi** un type d'objet énumérateur (cf. paragraphe précédent). Notons également que le compilateur fournit une implémentation d'`IDisposable` pour tous les objets énumérateurs.

Un objet énumérable doit en plus implémenter `IEnumerable` : il lui suffit de retourner un self pointer ou un clone au niveau de sa méthode `GetEnumerator`. Un objet énumérable de type `T` retournera donc une instance de type `T`.

Cela signifie que le type généré à la compilation pour un objet énumérateur possède les membres suivants :

- Méthode `MoveNext()`
- Propriété `Current`
- Méthode `Reset()`
- Méthode `Dispose()`

La méthode `MoveNext` est probablement la plus intéressante ici : le compilateur va faire en sorte de générer son corps de façon à ce qu'elle respecte la même logique d'itération que celle qui était décrite par le code du bloc itérateur original. Tout le challenge consiste à faire en sorte que les appels successifs à `MoveNext` contrôlent bien l'itération comme le code du bloc itérateur le spécifie. Cela implique que toutes les variables et objets sollicités au niveau du bloc itérateur par l'algorithme devront être transformés en membres d'instance de l'objet énumérateur pour que leur état soit conservé entre chaque appel.

Par exemple, si le bloc itérateur est le suivant :

```
for (int i = 1; i <= 10; i++)
{
    yield return i;
}
```

Le compilateur devra créer un type d'objet énumérateur possédant un champ d'instance (ex: `private int cpt;`) pour conserver l'état du compteur utilisé dans la boucle `for` du bloc itérateur. Lors de la création de l'objet énumérateur, son état est initialisé en fonction des valeurs initiales de toutes les variables sollicitées dans l'algorithme du bloc itérateur. Dans le cas précédent, le membre `cpt` sera initialisé avec la valeur 1.

Nous allons voir à présent comment faire la correspondance entre le code du bloc itérateur (celui que vous écrivez) et le code de la méthode `MoveNext` (généré par le compilateur). Nous allons pour cela considérer **même si c'est inexact** que le flux d'exécution se déplace dans le code du bloc itérateur à chaque fois que la méthode `MoveNext` est appelée sur l'objet énumérateur.

Lors du premier appel à la méthode `MoveNext`, le bloc itérateur est exécuté à partir de la première ligne jusqu'à ce que l'exécution soit interrompue par l'une des conditions suivantes :

- **L'expression yield return <expression>; est rencontrée** : La valeur de <expression> est affectée à la propriété Current de l'objet énumérateur, et toutes les valeurs des variables locales utilisées par le bloc itérateur sont sauvegardées. La méthode MoveNext retourne true.
- **L'expression yield break; est rencontrée** : Si une clause finally existait pour le code contenant le yield break, elle est exécutée. La méthode MoveNext retourne false.
- **La fin du bloc itérateur est rencontré** : La méthode MoveNext retourne false.
- **Une exception non capturée est levée** : Si une clause finally existe pour le code ayant levé l'exception, elle est exécutée. L'exception est propagée à l'appelant de la méthode MoveNext.

Lors des appels subséquents à la méthode MoveNext, la même logique est respectée à la différence près que l'exécution du bloc itérateur reprend **à la ligne suivant l'expression yield return ayant provoqué la sortie précédente de la méthode**. Il est important de savoir que l'état des variables locales ayant été sauvegardées lors de la sortie est restauré avant que l'exécution ne reprenne. Si la méthode MoveNext est appelée de nouveau alors qu'elle a déjà signifié la fin de l'itération en retournant false, elle continue à retourner false.

## 4 - Quelques exemples

### 4-1 - Nombres pairs

```
public IEnumerable GetValues(int limit)
{
    for (int i = 0; i < limit; i++)
    {
        if (i%2 == 0)
        {
            yield return i;
        }
    }
}

[Test]
public void Test()
{
    foreach (object o in GetValues(10))
    {
        Console.WriteLine(o);
    }
}
```

Affiche les 5 premiers nombres pairs : 0, 2, 4, 6, 8.

### 4.2 - Yield c'est rigolo

```
public IEnumerable GetValues()
{
    yield return "Yield";
    yield return " c'est";
    yield return " rigolo.";
}

[Test]
public void Test()
{
    foreach (object o in GetValues())
    {
        Console.Write(o);
    }
}
```

Affiche : "Yield c'est rigolo#. Plusieurs yield return peuvent se succéder dans le même bloc itérateur.

## 4.3 - Yield break

```

public IEnumerable GetFStrings(IEnumerable<string> strings)
{
    foreach (string s in strings)
    {
        if (s.ToUpper().StartsWith("F"))
        {
            yield return s;
        }
        else
        {
            yield break;
        }
    }
}

[Test]
public void Test()
{
    string[] strings = new string[]{"fern","fern2","fern3","notFern", "fern4"};
    foreach (object o in GetFStrings(strings))
    {
        Console.WriteLine(o);
    }
}
    
```

Affiche les chaînes de la collection qui commencent par la lettre "F". Si une chaîne de caractères ne commence pas par la lettre "F", l'inspection est stoppée. La méthode affiche donc: "fern", "fern2", "fern3".

## 4.4 - Valeurs calculées

```

public IEnumerable GetComputedValues(int[] toCompute)
{
    foreach (int i in toCompute)
    {
        Thread.Sleep(1000);
        yield return i*i;
    }
}

[Test]
public void Test()
{
    int[] toCompute = new int[]{12,21,40,3,78};
    foreach (int computedValue in GetComputedValues(toCompute))
    {
        if (computedValue > 150)
        {
            Console.WriteLine(computedValue);
            break;
        }
    }
}
    
```

Affiche la première valeur de l'énumération supérieure à 150, et stoppe l'itération : 441. (21<sup>2</sup>)

## 5 - Exécution différée et conséquences

Une des caractéristiques de ce mécanisme est aussi trompeuse qu'utile. Nous avons largement insisté sur le fait que le code du bloc itérateur n'était pas réellement exécuté, et qu'il permettait simplement au compilateur de générer

un objet énumérateur. Cela signifie qu'à l'appel de la méthode possédant le bloc itérateur, aucun code d'itération n'est exécuté.

Ce dernier est atteint lors du premier appel à la méthode MoveNext de l'objet énumérateur retourné par le compilateur, et continuera d'être exécuté au **fur et à mesure** de l'itération. Pour simplifier, et qualifier cette caractéristique, il est possible de parler d'**exécution différée du bloc itérateur**. Tentons de l'exploiter dans quelques exemples choisis :

La méthode GetComputedValues du dernier exemple peut être réécrite sans que le mot clé yield ne soit utilisé :

```
public IEnumerable GetComputedValues(int[] toCompute)
{
    List<int> computedValues = new List<int>();
    foreach (int i in toCompute)
    {
        Thread.Sleep(1000);
        computedValues.Add(i*i);
    }
    return computedValues;
}
```

La pause d'une seconde est uniquement là pour simuler une opération coûteuse en temps. Reprenons ensuite le même scénario d'itération, mais avec cette version de la méthode GetComputedValues :

```
[Test]
public void Test()
{
    int[] toCompute = new int[]{12,21,40,3,78,45,789,12,654,10};
    foreach (int computedValue in GetComputedValues(toCompute))
    {
        if (computedValue > 150)
        {
            Console.WriteLine(computedValue);
            break;
        }
    }
}
```

On constate que même si l'itération est stoppée dans le code client dès qu'une des valeurs dépasse 150, le calcul coûteux a été exécuté pour chacune des 10 valeurs de la liste d'arguments, portant le temps d'exécution de tout le scénario à environ 10 secondes. En utilisant la méthode GetComputedValues basée sur yield, on réduit le temps d'exécution du scénario à 2 secondes, puisqu'uniquement 2 calculs seront effectués.

Voici quelques autres exemples amusants :

La méthode suivante peut être appelée sans provoquer de boucle infinie, puisqu'elle est pseudo-exécutée au fur et à mesure. Il faut juste faire attention lors de l'itération, et prévoir un cas terminal au niveau du code client :

```
public IEnumerable Get42()
{
    while (true)
    {
        yield return 42;
    }
}
```

La code suivant ne provoque pas d'exception, puisque l'itération n'a pas lieu. L'exception serait déclenchée lors de la première itération, c'est à dire lors du premier appel à la méthode MoveNext de l'objet énumérateur :

```
public IEnumerable<char> GetChars(string s)
```

```

    {
        if (string.IsNullOrEmpty(s))
        {
            throw new ArgumentNullException("s");
        }

        foreach (char c in s)
        {
            yield return c;
        }
    }

[Test]
public void Test()
{
    IEnumerable<char> chars = GetChars(null);
}
    
```

L'appel à la méthode TestValues de la classe suivante provoque l'affichage : 0,1,2,3,4 et non pas 0,1,2,3,4,5,6,7,8,9. En effet, la logique d'énumération du bloc itérateur dépend de la valeur d'un membre d'instance de la classe FunnyYieldTest. A chaque appel de la méthode MoveNext sur l'objet énumérateur, le champ counter sera réévalué.

```

public class FunnyYieldTest
{
    private int counter;

    private IEnumerable<int> GetValues()
    {
        for (int i = 0; i < this.counter; i++)
        {
            yield return i;
        }
    }

    private void TestValues()
    {
        this.counter = 10;
        IEnumerable<int> values = GetValues();
        this.counter = 5;
        foreach (int i in values)
        {
            Console.WriteLine(i);
        }
    }
}
    
```

Les plus vicieux d'entre vous pourront même tenter de modifier la valeur du champ privé counter **durant** l'itération.

Bref, si vous avez l'habitude d'utiliser LINQ, cela doit vous parler. Une requête LINQ possède le même mode d'exécution différé, car elle s'appuie sur le mécanisme des blocs itérateurs de C# 2.0. Vous pouvez vous amuser à décompiler la classe Enumeration du namespace System.Linq, vous verrez que les méthodes d'extensions utilisent toutes yield. Enfin, le retro-engineering ne fera pas apparaître le mot clé yield directement, mais plutôt les classes que le compilateur a généré en interprétant les blocs itérateurs.

## 6 - Pour aller plus loin

Un bon moyen de comprendre ce qui se passe réellement lors de l'interprétation des blocs itérateurs par le compilateur consiste effectivement à inspecter le bytecode généré. Reflector permet même d'afficher le code C# correspondant au code IL, ce qui est assez pratique.

Reprenons la méthode GetFirstTenIntegers :

```


```

```
public class YieldAutopsy
{
    public IEnumerable<int> GetFirstTenIntegers()
    {
        for (int i = 1; i <= 10; i++)
        {
            yield return i;
        }
    }
}
```

Le compilateur va produire le code IL correspondant à :

```
public IEnumerable<int> GetFirstTenIntegers()
{
    <GetFirstTenIntegers>d__4 d__ = new <GetFirstTenIntegers>d__4(-2);
    d__.<>4__this = this;
    return d__;
}
```

Il s'agit de la déclaration, de l'instanciation, de l'initialisation et du retour de l'objet énumérable. Le type de l'objet énumérable généré par le compilateur possède un nom barbare : <GetFirstTenIntegers>d\_\_4. La présence des caractères interdits dans l'identifiant permettent d'éviter tout conflit avec un type éventuellement existant dans votre projet.

Je ne vais pas vous copier ici tout le code de cette classe générée, mais en voici tout de même la déclaration :

```
[CompilerGenerated]
private sealed class <GetFirstTenIntegers>d__0 : IEnumerable<int>,
    IEnumerable,
    IEnumerator<int>,
    IEnumerator,
    IDisposable
{
    // Membres.
}
```

On constate qu'elle implémente IEnumerable et IEnumerator, ainsi qu'IDisposable. Si la méthode GetFirstTenIntegers renvoyait un IEnumerator, la classe générée le compilateur aurait simplement implémenté IEnumerator et IDisposable. Il s'agit donc là d'un objet énumérable.

Lorsque les deux interfaces sus citées sont implémentées par la même classe, IEnumerator est souvent implémenté **explicitement**. C'est le cas ici, et cela permet d'implémenter IEnumerable de la façon suivante tout en cachant les membres d'IEnumerator aux portions du code client manipulant la classe en tant qu'IEnumerable :

```
public IEnumerator GetEnumerator()
{
    return (IEnumerator)this;
}
```

Il va falloir me faire confiance : la classe générée par le compilateur respecte le même pattern, à la différence près qu'elle doit également gérer le fait d'implémenter les versions génériques de IEnumerable et IEnumerator.

Ce qui nous intéresse maintenant est de voir comment le bloc itérateur a été interprété par le compilateur, et quel est le résultat concret de cette interprétation au niveau de la méthode MoveNext :

```
private bool MoveNext()
{
    switch (this.<>1__state)
    {
```

```

    {
        case 0:
            this.<>1__state = -1;
            this.<i>5__1 = 1;
            while (this.<i>5__1 <= 10)
            {
                this.<>2__current = this.<i>5__1;
                this.<>1__state = 1;
                return true;
            }
            Label_0046:
                this.<>1__state = -1;
                this.<i>5__1++;
            }
            break;

        case 1:
            goto Label_0046;
    }
    return false;
}
    
```

Ce que vous devez savoir pour comprendre ce code :

- <>1\_\_state est un membre privé de type entier, égal à 0 lors du premier appel de la méthode.
- <i>5\_\_1 est un membre privé de type entier.
- <>2\_\_current est un membre privé de type entier, qui est accédé via les propriétés IEnumerator.Current et IEnumerator<int>.Current de la classe.
- Les goto, c'est le mal. Le compilateur s'en sert parce qu'il est moins faillible que nous, et c'est très bien comme ça. D'ailleurs, même s'il est possible d'utiliser goto en C#, certaines limitations nous empêchent de recourir à des algorithmes tels que celui qui est généré ici. Bien que le code IL de ce dernier soit valide, le compilateur C# ne sera pas capable de détecter le label Label\_0046 si nous essayons d'écrire directement en C# le code précédent. Il lèvera l'erreur de génération CS0159 : "*Il n'existe pas d'étiquette #Label\_0046' dans la portée de l'instruction goto*".

On constate donc que le code présent dans la méthode MoveNext a pour but de contrôler l'itération exactement comme cela a été spécifié au niveau du bloc itérateur, en utilisant le mot clé yield. Cette transformation est relativement complexe, puisque le compilateur est capable de convertir l'algorithme original pour produire une sorte de **machine à états** au niveau de la classe générée, pouvant supporter l'exécution différée grâce à la persistance des valeurs des différentes variables nécessaires au contrôle de l'itération entre chaque appel à MoveNext.

Ici, il s'agissait de traduire une simple boucle for en conservant l'état du compteur, mais certains blocs itérateurs peuvent être éminemment plus compliqués. Prenons juste la méthode GetFStrings de la section d'exemple, et inspectons le code que le compilateur produit pour la méthode MoveNext de l'itérateur :

```

private bool MoveNext()
{
    try
    {
        switch (this.<>1__state)
        {
            case 0:
                this.<>1__state = -1;
                this.<>7__wrap2 = this.strings.GetEnumerator();
                this.<>1__state = 1;
                while (this.<>7__wrap2.MoveNext())
                {
                    this.<s>5__1 = this.<>7__wrap2.Current;
                    if (!this.<s>5__1.ToUpper().StartsWith("F"))
                    {
                        goto Label_0097;
                    }
                    this.<>2__current = this.<s>5__1;
                    this.<>1__state = 2;
                }
            }
        }
    }
}
    
```

```

        return true;
Label_008D:
        this.<>1__state = 1;
        goto Label_00A1;
Label_0097:
        ((IDisposable) this).Dispose();
        break;
Label_00A1:
    }
    this.<>1__state = -1;
    if (this.<>7__wrap2 != null)
    {
        this.<>7__wrap2.Dispose();
    }
    break;

    case 2:
        goto Label_008D;
    }
    return false;
}
fault
{
    ((IDisposable) this).Dispose();
}
}

```

On constate que l'itérateur généré possède une référence vers la collection utilisée dans le bloc itérateur (`this.strings`) et que l'implémentation de la méthode `MoveNext` doit faire appel à l'énumérateur de cette même collection. Rien n'empêche cette dernière d'utiliser un bloc itérateur comme implémentation de sa méthode `GetEnumerator`.

C'est d'ailleurs le cas avec LINQ, puisque son API repose sur une **Fluent Interface**. La requête suivante :

```

IEnumerable<string> query = from s in names
                           where s.Length == 5
                           orderby s
                           select s.ToUpper();

```

Peut être écrite également :

```

IEnumerable<string> query = names
    .Where(s => s.Length == 5)
    .OrderBy(s => s)
    .Select(s => s.ToUpper());

```

Où les méthodes `Where`, `OrderBy` et `Select` retournent des objets énumérables. La logique complète et **réelle** d'exécution d'une telle requête peut vite atteindre une complexité vertigineuse, car les objets énumérables vont se solliciter entre eux. En parallèle, l'écriture des requêtes est quasi-enfantine, tandis que l'implémentation des différents blocs itérateurs reste aisée. C'est assez révélateur de la puissance des itérateurs de C#.

## 7 - Quelques restrictions

- `yield` ne peut pas être utilisé dans un contexte `unsafe`.
- `yield` ne peut pas être utilisé dans une méthode anonyme ou une expression `lambda`.
- Les blocs itérateurs ne peuvent être utilisés dans les méthodes acceptant des paramètres `out` et/ou `ref`.
- La méthode `Reset` des objets énumérateurs n'est jamais implémentée par le compilateur C# :

```

[DebuggerHidden]
void IEnumerator.Reset()

```

```
{  
    throw new NotSupportedException();  
}
```

Il est donc particulièrement dangereux d'écrire :

```
IEnumerable<int> firstIntegers = GetFirstTenIntegers();  
IEnumerator<int> enumerator = firstIntegers.GetEnumerator();  
enumerator.Reset();
```

## 8 - Conclusion

Les blocs itérateurs en C#, c'est de la magie sans être de la magie. On entend quelque fois certains détracteurs de Microsoft expliquer que C#, notamment dans les nouvelles versions, se permet de bousculer les paradigmes empiriques de l'OOP avec les itérateurs, les expressions lambda, les méthodes d'extension, LINQ, etc. Je pense que lorsqu'on prend la peine de comprendre comment ces solutions sont implémentées, on se rend compte qu'elles n'ont pour but que de rendre la vie des gens qui font de l'ordinateur plus simple. En général, il s'agit d'abstractions ou de surcouches dont les mécanismes sont basés sur les concepts fondamentaux de l'Objet.

J'aurais probablement l'occasion d'en reparler, mais les itérateurs en C#2.0 ou des méthodes d'extensions en C# 3.0 ne sont que des timesavers légitimes. Dans le premier cas, on délègue au compilateur une tâche pénible qu'il est parfaitement possible de définir en utilisant un enrichissement mineur du langage. Dans le second, on offre juste une façon supplémentaire d'appeler certaines méthodes statiques encapsulant un traitement relatif à un type d'objet. Etc.