Casse-têtes en C# (réponses)

par Jon Skeet (Page d'accueil) (Blog) Jean-Michel Ormes (traduction) (Page d'accueil) (Blog)

Date de publication : 11/04/2012

Dernière mise à jour :

Voici les réponses aux **casse-têtes en C#**. J'ai inclus les questions à nouveau pour un peu plus de clarté.

Traduction

Ceci est la traduction la plus fidèle possible de l'article de Jon Skeet, **C# Brainteasers**.

II - Surcharge

Qu'est-ce qui est affiché, et pourquoi ?

```
using System;

class Base
{
    public virtual void Foo(int x)
    {
        Console.WriteLine ("Base.Foo(int)");
    }
}

class Derived : Base
{
    public override void Foo(int x)
    {
        Console.WriteLine ("Derived.Foo(int)");
    }

    public void Foo(object o)
    {
        Console.WriteLine ("Derived.Foo(object)");
    }
}

class Test
{
    static void Main()
    {
        Derived d = new Derived();
        int i = 10;
        d.Foo(i);
    }
}
```

Réponse : Derived.Foo(object) est affiché. Lorsque l'on choisit une surcharge, s'il y a des méthodes compatibles *déclarées* dans une classe dérivée, toutes les signatures *déclarées* dans la classe de base sont ignorées, même si elles sont substitutées dans la même classe dérivée !

III - Ordre ! Ordre !

Qu'est-ce qui va s'afficher, pourquoi, et en êtes-vous sûr ?

```
using System;

class Foo
{
    static Foo()
    {
        Console.WriteLine ("Foo");
    }
}

class Bar
{
    static int i = Init();
```

```
    static int Init()
    {
        Console.WriteLine("Bar");
        return 0;
    }
}

class Test
{
    static void Main()
    {
        Foo f = new Foo();
        Bar b = new Bar();
    }
}
```

Réponse : sur ma machine, Bar est affiché, puis Foo. En fait, Foo contient un constructeur statique qui ne peut être exécuté tant que le point exact où la classe est initialisée n'est pas atteint. Bar n'a pas de constructeur statique, donc le CLR peut l'initialiser plus tôt. Cependant, il n'y a rien qui garantit que Bar soit affiché. Aucun champ statique n'est référencé, donc en théorie, le CLR n'a pas à l'initialiser dans notre exemple. Tout cela est dû au **flag beforefieldinit**.

IV - Bête arithmétique

Les ordinateurs sont censés être bons en calcul, n'est-ce pas ? Alors pourquoi la console renvoie-t-elle "False" ?

```
double d1 = 1.000001;
double d2 = 0.000001;
Console.WriteLine((d1-d2)==1.0);
```

Réponse : toutes les valeurs ici sont stockées sous forme binaire à virgule flottante. Alors que 1.0 peut être stocké convenablement, 1.000001 est stocké ainsi : 1.000000999999999917733362053695600479841232299804687 5, et 0.000001 est stocké ainsi : 0.000000999999999999999954748111825886258685613938723690807819366455078125. La différence entre ces deux valeurs ne correspond pas à 1.0, et de ce fait la différence ne peut être correctement stockée. **En apprendre plus sur les nombres à virgule flottante en binaire.**

V - Print, print, print...

Voici un code utilisant la fonctionnalité de méthode anonyme de C# 2. Que fait-il ?

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

class Test
{
    delegate void Printer();

    static void Main()
    {
        List<Printer> printers = new List<Printer>();
        for (int i=0; i < 10; i++)
        {
            printers.Add(delegate { Console.WriteLine(i); });
        }

        foreach (Printer printer in printers)
        {
            printer();
        }
    }
}
```

Réponse : ah, les joies des **variables capturées** ! Il n'y a qu'une seule variable **i** ici, et sa valeur change à chaque itération de la boucle. Les méthodes anonymes récupèrent la variable elle-même plutôt que sa valeur au moment de la création, de sorte que le résultat 10 est imprimé dix fois !

VI - Il n'y a littéralement aucun problème avec le compilateur ici...

Est-ce que ce code pourrait compiler ? Compile-t-il ? Qu'est-ce que cela signifie ?

```
using System;

class Test
{
    enum Foo { Bar, Baz };

    static void Main()
    {
        Foo f = 0.0;
        Console.WriteLine(f);
    }
}
```

Réponse : cela ne devrait pas compiler, mais il le fait avec les compilateurs MS aussi bien pour C# 2 que pour C# 3 (et probablement 1 aussi, je n'ai pas vérifié). Il ne devrait pas compiler parce que seul le littéral 0 devrait être implicitement convertible en la valeur par défaut de n'importe quelle énumération. Ici, la valeur décimale est 0,0. Juste un petit bogue du compilateur. Le résultat est que cela affiche Bar puisque c'est la valeur par défaut de Foo.

En voici d'autres dans le même genre...

```
using System;

class Test
{
    enum Foo { Bar, Baz };

    const int One = 1;
    const int Une = 1;

    static void Main()
    {
        Foo f = One-Une;
        Console.WriteLine(f);
    }
}
```

Réponse : Erf !! C'est de pire en pire ! Ce code ne compilera pas pour C# 2 mais le *fera* pour C# 3. C'est un bogue connu dû à une optimisation qui s'effectue trop tôt : la collecte de constantes de valeur 0 et le fait de penser que toute constante 0 connue puisse être convertible à la valeur 0 de n'importe quelle énumération. Ce bogue ne sera probablement jamais corrigé car cela pourrait "casser" du code existant qui est techniquement illégal, mais qui fonctionne parfaitement bien. Il est toutefois possible que ce soient les spécifications qui changent.

VII - Inférence de type à gogo

J'ai d'abord vu ceci sur **le blog d'Ayende** (sous une forme un peu plus obscure, il est vrai). Encore une fois, réfléchissez sur ce que le code va afficher et pourquoi.

```
using System;

class Test
{
```

```
    static void Main()
    {
        Foo("Hello");
    }

    static void Foo(object x)
    {
        Console.WriteLine("object");
    }

    static void Foo<T>(params T[] x)
    {
        Console.WriteLine("params T[]");
    }
}
```

Réponse : params T [] est affiché. Mais pourquoi le compilateur choisirait-il de créer un tableau alors qu'il n'en a pas besoin ? Eh bien... il y a deux étapes qui expliquent cela. Tout d'abord, en essayant de trouver les surcharges qui sont des candidates légitimes à être appelées, l'inférence de type détermine que T devrait être de type System.String. Rien d'effrayant à ce jour.

Puis les surcharges essaient de travailler avec la méthode qui semble être la plus «intéressante». Si c'est un choix entre string x et params string [] x, le premier gagnera toujours - mais à ce stade, c'est *effectivement* un choix entre object x et params string [] x. (Le fait que ce soit une méthode générique n'est pertinent que dans une situation d'égalité.) Dans l'optique de trouver de "meilleures conversions", la *forme développée* de la méthode avec le paramètre params est ensuite utilisée. Cela signifie que lorsque les conversions sont étudiées, le choix est entre object x ou string x - très clairement, ce dernier l'emporte.

VIII - Remerciements

Je tiens à remercier Jon Skeet pour son aimable autorisation de traduire cet article, ainsi que **tomlev** et **ClaudeLELOUP** pour la relecture attentive et les corrections apportées.