C# : généricité

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

C#

Concept objet

- Classe = { attributs } + { méthodes }
- **C#** : langage fortement typé $\Rightarrow$ Les attributs ont **forcément** un nom et un type
- Le type d'un attribut est fixé par la définition de la classe et est identique pour toutes ses instances.
- Les méthodes permettent généralement d'effectuer des opérations sur les attributs tout en respectant leurs spécificités (type, taille...)

C#

Problématique

- Et si on a besoin d'une classe dont les méthodes effectuent les mêmes opérations quel que soit le type d'attributs
  - **somme** pour **entiers** ou **réels**,
  - **concaténation** pour **chaînes de caractères**,
  - **ou logique** pour **booléens**...
  - ...

C#

Deux solutions

- Dupliquer les classes : peu maintenable
- Utiliser `object` + cast : erreurs à l'exécution, boxing/unboxing

C#

Une solution plus élégante avec la généricité

- Ne pas fixer le type concrètement à la définition de la classe, mais le paramétrer via un type générique.
- À l'instanciation de la classe, on précise le type à utiliser par cette classe pour cette instance.
- **On peut donc choisir pour chaque instance le type que l'on souhaite utiliser.**

C#

Généricité

La généricité permet de définir une classe, une méthode ou une interface paramétrée par un ou plusieurs types, fournis **à l'utilisation** (et vérifiés à la compilation).

- Typage fort conservé
- Code réutilisable
- Généralement plus performant que `object`

C#

Une première classe avec un type générique

```
namespace CoursPoo
{
    internal class Exemple<T>
    {
        public T Var { get; set; }
    }
}
```

C#

Une première classe avec un type générique

```
namespace CoursPoo
{
    internal class Exemple<T>
    {
        public T Var { get; set; }
    }
}
```


Explication

- `T` est un paramètre de type.
- Le type concret est fourni à l'instanciation : `Exemple<int>`, `Exemple<string>`, etc.

C#

Créons des instances de la même classe avec des types différents

```
Exemple<int> entier = new Exemple<int>();
entier.Var = 10;
Console.WriteLine(entier.Var.GetType().Name + " " + entier.Var);
Int32 10

Exemple<string> chaine = new Exemple<string>();
chaine.Var ="Bonjour";
Console.WriteLine(chaine.Var.GetType().Name + " " + chaine.Var);
// String Bonjour
```

C#

Créons des instances de la même classe avec des types différents

```
Exemple<int> entier = new Exemple<int>();
entier.Var = 10;
Console.WriteLine(entier.Var.GetType().Name + " " + entier.Var);
Int32 10

Exemple<string> chaine = new Exemple<string>();
chaine.Var ="Bonjour";
Console.WriteLine(chaine.Var.GetType().Name + " " + chaine.Var);
// String Bonjour
```


Point clé

`Exemple<int>` et `Exemple<string>` sont des types distincts du point de vue du compilateur.

C#

Considérons la classe `Operation` acceptant un type générique pour les deux propriétés `Var1` et `Var2`

```
internal class Operation<T>
{
    public T Var1 { get; set; }
    public T Var2 { get; set; }

    public Operation(T var1, T var2)
    {
        Var1 = var1;
        Var2 = var2;
    }
}
```

C#

Exercice

Définissez une méthode `Plus` qui affiche

- la somme si les deux propriétés sont de type `Double` ou `Int32`,
- la concaténation si les deux propriétés sont de type `String`,
- le résultat du **ou logique** si les deux propriétés sont de type `Boolean`,
- un message `Type non traité par notre méthode` pour tous les autres types.

C#

Solution

```
public void Plus()
{
    var resultat = (MyProperty1, MyProperty2) switch
    {
        (int i1, int i2) => (i1 + i2).ToString(),
        (double i1, double i2) => (i1 + i2).ToString(),
        (bool b1, bool b2) => (b1 || b2).ToString(),
        (string s1, string s2) => s1 + s2,
        _ => "Type non supporté"
    };
    Console.WriteLine(resultat);
}
```

C#

Pour tester

```
namespace CoursPoo
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Operation<int> operation1 = new Operation<int>(5, 3);
            operation1.Plus();
            Operation<String> operation2 = new Operation<String>("bon", "jour");
            operation2.Plus();
            Operation<Double> operation3 = new Operation<Double>(5.2, 3.8);
            operation3.Plus();
            Operation<Boolean> operation4 = new Operation<Boolean>(true, false);
            operation4.Plus();
        }
    }
}
```

C#

Pour tester

```
namespace CoursPoo
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Operation<int> operation1 = new Operation<int>(5, 3);
            operation1.Plus();
            Operation<String> operation2 = new Operation<String>("bon", "jour");
            operation2.Plus();
            Operation<Double> operation3 = new Operation<Double>(5.2, 3.8);
            operation3.Plus();
            Operation<Boolean> operation4 = new Operation<Boolean>(true, false);
            operation4.Plus();
        }
    }
}
```

Résultat

```
8
bonjour
9
True
```

C#

Bonne pratique : utiliser la contrainte `where` quand c'est possible [C# 11 : .NET 7]

```
using System.Numerics;

class Operation<T> where T : INumber<T>
{
    public T Var1 { get; set; }
    public T Var2 { get; set; }

    public void Plus()
    {
        Console.WriteLine(Var1 + Var2);
    }
}
```

C#

Ainsi, les seuls types autorisés sont les types numériques

```
namespace CoursPoo
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Operation<int> operation1 = new Operation<int>(5, 3);
            operation1.Plus();
            Operation<Double> operation3 = new Operation<Double>(5.2, 3.8);
            operation3.Plus();

        }
    }
}
```

C#

Ainsi, les seuls types autorisés sont les types numériques

```
namespace CoursPoo
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Operation<int> operation1 = new Operation<int>(5, 3);
            operation1.Plus();
            Operation<Double> operation3 = new Operation<Double>(5.2, 3.8);
            operation3.Plus();

        }
    }
}
```


Résultat

```
8
9
```

C#

Exemple avec deux types génériques

```
namespace CoursPoo
{
    internal class Exemple2<T, S>
    {
        public T Var1 { get; set; }
        public S Var2 { get; set; }
    }
}
```

C#

Testons tout ça

```
namespace CoursPoo
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Exemple2<int, string> couple = new();
            couple.Var1 = 10;
            couple.Var2 = "Bonjour";
            Console.WriteLine(couple.Var1.GetType().Name + " " + couple.Var1);
            Console.WriteLine(couple.Var2.GetType().Name + " " + couple.Var2);
        }
    }
}
```

C#

Testons tout ça

```
namespace CoursPoo
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Exemple2<int, string> couple = new();
            couple.Var1 = 10;
            couple.Var2 = "Bonjour";
            Console.WriteLine(couple.Var1.GetType().Name + " " + couple.Var1);
            Console.WriteLine(couple.Var2.GetType().Name + " " + couple.Var2);
        }
    }
}
```


Résultat

```
Int32 10
String Bonjour
```

C#

La généricité autorise les types nullables, les classes, les structs...

```
namespace CoursPoo
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Exemple2<int?, string> couple = new();
            couple.Var1 = 2;
            couple.Var2 = "Bonjour";
            Console.WriteLine(couple.Var1.GetType().Name + " " + couple.Var1);
            Console.WriteLine(couple.Var2.GetType().Name + " " + couple.Var2);
        }
    }
}
```

C#

Pour autoriser uniquement les classes, on peut utiliser les contraintes

```
namespace CoursPoo
{
    class Exemple2<T, S> where T : class
    {
        public T Var1 { get; set; }
        public S Var2 { get; set; }
    }
}
```

C#

L'écriture précédente est soulignée en rouge

```
namespace CoursPoo
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Exemple2<int, string> couple = new();
            couple.Var1 = 2;
            couple.Var2 = "Bonjour";
            Console.WriteLine(couple.Var1.GetType().Name + " " + couple.Var1);
            Console.WriteLine(couple.Var2.GetType().Name + " " + couple.Var2);
        }
    }
}
```

C#

Quelques autres contraintes prédéfinies

- `where T : class` (type référence)
- `where T : struct` (type valeur)
- `where T : class?` (disponible lorsque les types de référence nullables (Nullable Reference Types) sont activés (à partir de **C# 8.0**))
- `where T : notnull` (interdit null)
- `where T : new()` (doit avoir un constructeur public sans paramètre)

C#

Remarques

- Les paramètres de type générique doivent être préfixés par `T` (par exemple, `T` pour `Type`, `TKey` et `TValue` pour les dictionnaires...).
- Il est possible d'appliquer plusieurs contraintes à un seul paramètre de type générique, en les séparant par des virgules (par exemple, `where T : class, IDisposable, new()`).
- Les méthodes et les interfaces aussi peuvent être génériques.

C#

Exemple d'une méthode utilisant un type générique

```
public static void Swap<T>(ref T a, ref T b)
{
    T tmp = a;
    a = b;
    b = tmp;
}
```

C#

Exemple d'une méthode utilisant un type générique

```
public static void Swap<T>(ref T a, ref T b)
{
    T tmp = a;
    a = b;
    b = tmp;
}
```

Appel

```
int a = 5;
int b = 10;
Console.WriteLine($"Avant swap: a={a}, b={b}");
//  Avant swap: a=5, b=10

Swap<int>(ref a, ref b);

Console.WriteLine($"Après swap: a={a}, b={b}");
//  Après swap: a=10, b=5
```

C#

À l'appel, le type générique peut être omis et il sera automatiquement déduit

```
int a = 5;
int b = 10;
Console.WriteLine($"Avant swap: a={a}, b={b}");
//  Avant swap: a=5, b=10

Swap(ref a, ref b);

Console.WriteLine($"Après swap: a={a}, b={b}");
//  Après swap: a=10, b=5
```