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 ⇒ Les attributs ont **forcément** un nom et un type
- Le type d'attribut ne change pas pour toutes les instances d'une classe
- 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**...
  - ...
- **Impossible sans définir plusieurs classes (une pour chaque type) et une interface ou en imposant le type `Object` et en utilisant plusieurs cast...**

C#

Une solution plus élégante avec la généricité

- Ne pas définir de type pour les attributs
- Définir un type générique qui ne sera pas précisé à la création de la classe
- À l'instanciation de la classe, on précise le type à utiliser par cette classe
- **On peut donc choisir pour chaque instance le type que l'on souhaite utiliser.**

C#

Une première classe avec un type générique

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

C#

Créons des instances de la même classe avec des types différents

```
namespace CoursPoo
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Exemple<int> entier = new Exemple<int>();
            entier.Var = 10;
            Console.WriteLine(entier.Var.GetType().Name + " " + entier.Var);
            Exemple<string> chaine = new Exemple<string>();
            chaine.Var ="Bonjour";
            Console.WriteLine(chaine.Var.GetType().Name + " " + chaine.Var);
        }
    }
}
```

C#

Créons des instances de la même classe avec des types différents

```
namespace CoursPoo
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Exemple<int> entier = new Exemple<int>();
            entier.Var = 10;
            Console.WriteLine(entier.Var.GetType().Name + " " + entier.Var);
            Exemple<string> chaine = new Exemple<string>();
            chaine.Var ="Bonjour";
            Console.WriteLine(chaine.Var.GetType().Name + " " + chaine.Var);
        }
    }
}
```

Résultat

```
Int32 10
String Bonjour
```

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#

Première solution avec `GetType`

```
public void Plus()
{
    if (Var1.GetType().Name.Equals("Double"))
    {
        Console.WriteLine(double.Parse(Var1.ToString()) + double.Parse(Var2.ToString()));
    }
    else if (Var1.GetType().Name.Equals("Int32"))
    {
        Console.WriteLine(int.Parse(Var1.ToString()) + int.Parse(Var2.ToString()));
    }
    else if (Var1.GetType().Name.Equals("Boolean"))
    {
        Console.WriteLine(bool.Parse(Var1.ToString()) || bool.Parse(Var2.ToString()));
    }
    else if (Var1.GetType().Name.Equals("String"))
    {
        Console.WriteLine(Var1.ToString() + Var2.ToString());
    }
    else
    {
        Console.WriteLine("Type non traité par notre méthode");
    }
}
```

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#

Deuxième solution avec `typeof`

```
public void Plus()
{
    var type = typeof(T).Name;
    if (type.Equals("Double"))
    {
        Console.WriteLine(double.Parse(Var1.ToString()) + double.Parse(Var2.ToString()));
    }
    else if (type.Equals("Int32"))
    {
        Console.WriteLine(int.Parse(Var1.ToString()) + int.Parse(Var2.ToString()));
    }
    else if (type.Equals("Boolean"))
    {
        Console.WriteLine(bool.Parse(Var1.ToString()) || bool.Parse(Var2.ToString()));
    }
    else if (type.Equals("String"))
    {
        Console.WriteLine(Var1.ToString() + Var2.ToString());
    }
    else
    {
        Console.WriteLine("Type non traité par notre méthode");
    }
}
```

C#

Troisième solution avec `is` [C# 7.0]

```
public void Plus()
{
    if (Var1 is int d1 && Var2 is int d2)
    {
        Console.WriteLine(d1 + d2);
    }
    else if (Var1 is int i1 && Var2 is int i2)
    {
        Console.WriteLine(i1 + i2);
    }
    else if (Var1 is bool b1 && Var2 is bool b2)
    {
        Console.WriteLine(b1 || b2);
    }
    else if (Var1 is string s1 && Var2 is string s2)
    {
        Console.WriteLine(s1 + s2);
    }
    else
    {
        Console.WriteLine("Type non traité par notre méthode");
    }
}
```

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 Exemple2<int, string>();
            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 Exemple2<int, string>();
            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 Exemple2<int?, string>();
            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 : classes
    {
        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 Exemple2<int?, string>();
            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`
- `where T : struct`
- `where T : class?`
- `where T : notnull`
- ...