ISEN ALL IS DIGITAL! LILLE | yncréa

Cours de C#

DENELE Lucas
FONTAINE Maxime

TABBARA Arnaud
CECCOTTI Romain

ISEN Lille | M1 | 2018-2019

Les Génériques : Qu'est-ce que c'est ?

Une classe qui permet de définir de nouvelles classes, ou méthodes dont **le type** (Integer, String, ...) devient un **paramètre**.

Avantages

- Type-Safety : S'assure que chaque élément possède le même type et permet « l'intelliSense » ce qui détecte et indique les erreurs en direct lors de l'écriture du code.

- Meilleur performance et code de qualité, car cela permet de réutiliser des algorithmes de process de données sans recopier le code spécifique au type.

- Evite le « boxing et unboxing ». *Expliqué un peu plus tard.*

- Réduit le nombre de cast nécessaires et évite des erreurs concernant ce point.

Inconvénient

- Complexité : Cela rajoute une nouvelle couche d'abstraction et de niveau de paramétrage en plus de ceux existants. Peut-être utilisé inutilement ou mal compris.

Les génériques dans les collections

Les collections non génériques : ArrayList, Hashtable, SortedList, Queue, ...

Les collections génériques : List<T>, Dictionary <T, U>, SortedList <T, U>, Queue <T>, ...

Quelle différence?

Pour des collections normales, dans le cas de value types (int, double, bool, char, etc.) il faut qu'ils soient 'boxed' dans un objet puis 'unboxed' pour être réutilisés plus tard ce qui utilise beaucoup de performance.

Les collections génériques permettent de reporter le moment de spécification du type jusqu'au moment de la création. Et vont ensuite les stocker dans des tableaux spécifiques au type concerné dont pas besoin de boxing ou de casting.

Créer une classe générique

Définition

Program.cs*

Test

```
1   using System;
2   // We use < > to specify Parameter type
    4 références
3   public class Test<T>
4   {
5       private T data;
        4 références
6       public T value
7       {
8           get
9           {
10              return this.data;
11          }
12          set
13          {
14              this.data = value;
15          }
16      }
17  }
18
```

Créer une classe générique

Main

```
22      static void Main(string[] args)
23          {
24
25              Test<string> name = new Test<string>();
26              name.value = "Class of C#";
27
28              // Affiche Class of C#
29              Console.WriteLine(name.value);
30
31
32              Test<float> version = new Test<float>();
33              version.value = 5.0F;
34
35              // Affiche 5
36              Console.WriteLine(version.value);
37          }
38      }
39
```

Créer une méthode générique

Program.cs*

Test

```
using System;

2 références
public class Test
{
    3 références
    public void Afficher<T>(string msg, T value)
    {
        Console.WriteLine("{0} : {1}", msg, value);
    }
}

0 références
public class Program
{
    // Main Method
    0 références
    public static int Main()
    {
        // on crée un objet de la classe Test.
        Test p = new Test();

        // On appelle la methode

        p.Afficher<int>("Integer", 122);
        // Affiche Integer : 122

        p.Afficher<char>("Character", 'H');
        //Affiche Character : H

        p.Afficher<double>("Decimal", 255.67);
        //Affiche Decimal : 255.67
        return 0;
    }
}
```

Exercice d'exemple

Créer une classe d'individu puis recréer une méthode générique « Where » similaire à la méthode Where de LINQ ci-dessous:

Création de la classe:

```
9 références
public class Individu
{
    6 références
    public string Job { get; set; }
    7 références
    public string Genre { get; set; }
    7 références
    public string Prenom { get; set; }
    7 références
    public int Age { get; set; }
}
```

Création d'une liste:

```
List<Individu> individus = new List<Individu>
{
    new Individu { Age = 22, Genre = "Masculin", Prenom = "Nicolas", Job = "patissier" },
    new Individu { Age = 36, Genre = "Feminin", Prenom = "Christine", Job = "professeur" },
    new Individu { Age = 67, Genre = "Masculin", Prenom = "Louis", Job = "medecin" },
    new Individu { Age = 10, Genre = "Masuclin", Prenom = "Tom", Job = "ecolier" },
    new Individu { Age = 41, Genre = "Masculin", Prenom = "Thomas", Job = "ingenieur" },
    new Individu { Age = 76, Genre = "Feminin", Prenom = "Camille", Job = "retraitee" }
};
```

Méthode Where à refaire:

```
var hommeMajeur = individus.Where<Individu>(individu => individu.Genre == "Masculin" && individu.Age>18 );
foreach (Individu individu in hommeMajeur)
{
    Console.WriteLine(individu.Prenom);
}
/*
Affiche :
Nicolas
Louis
Thomas */
```

1ère étape: déclaration de la méthode

```
public static class IEnumerableExtension
{
    public static IEnumerable<T> Where<T>(this IEnumerable<T> source, Func<T, bool> predicate)
    {
```

- *IEnumerable<T>* est une interface dont toutes les listes génériques de *C#* héritent. Il faut donc que notre méthode en soit une extension.
- Le <T> indique la généricité (T est une convention)
- Après le nom de la méthode nous indiquons entre chevrons tous les types génériques que cette méthode va devoir gérer (ici il n'y en a qu'un)
- « this » parce que le premier argument est l'objet lui-même auquel on appliquera notre méthode, et sera donc de type IEnumerable <T> générique
- Et le dernier argument est un delegate prenant un objet de type T et renvoyant un booléen. C'est la fonction qui contient la condition de renvoi des éléments.

2ème étape: contenu de la méthode

Notre fonction doit parcourir tous les objets de la collection, tester une condition sur chacun de ces objets et les inclure dans le résultat renvoyé si la condition est respectée. La condition qui elle est contenue dans le delegate, deuxième argument de notre méthode.

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> source, Func<T, bool> predicate)
{
    foreach (T obj in source)
    {
        if (predicate(obj))
        {
            yield return obj;
        }
    }
}
```

- Une boucle foreach pour parcourir toute la collection,
- On donne le nom « obj » a notre variable qui est du type « T » le type de la collection,
- in « source » qui correspond bien au nom de l'objet auquel on applique notre méthode (this) c'est-à-dire la collection,
- if (predicate(obj)) si l'objet passé dans le delegate renvoie true c'est-à-dire répond à la condition voulue,
- On retourne l'objet en question et ce pour tous les objets répondant à la condition.

Test de la méthode

On demande les individus dont le sexe est féminin et dont le prénom commence par C. Cela renvoie Christine et Camille, comme prévu. Notre méthode fonctionne comme prévu.

```
0 références
static void Main(string[] args)
{

    List<Individu> individus = new List<Individu>
    {
        new Individu { Age = 22, Genre = "Masculin", Prenom = "Nicolas", Job = "patissier" },
        new Individu { Age = 36, Genre = "Feminin", Prenom = "Christine", Job = "professeur" },
        new Individu { Age = 67, Genre = "Masculin", Prenom = "Louis", Job = "medecin" },
        new Individu { Age = 10, Genre = "Masuclin", Prenom = "Tom", Job = "ecolier" },
        new Individu { Age = 41, Genre = "Masculin", Prenom = "Thomas", Job = "ingenieur" },
        new Individu { Age = 76, Genre = "Feminin", Prenom = "Camille", Job = "retraitee" }
    };

    var Femme_Prenom_C = individus.Where<Individu>(individu => individu.Genre == "Feminin" && individu.Prenom.StartsWith("C") );
    foreach (Individu individu in Femme_Prenom_C)
    {
        Console.WriteLine(individu.Prenom);
    }
    Console.ReadLine();
    /*
    Affiche :
    Christine
    Camille */
```

Conclusion

Les génériques sont une des plus importantes nouveautés en *C#*. Ils permettent de résoudre un problème assez ennuyant auquel n'importe quel programmeur a déjà été confronté : la généricité. Ils ont de nombreux avantages et peu d'inconvénients et seront indispensables si l'on veut coder quelque chose d'assez développé en C#.