Cours de C#

DENELE Lucas TABBARA Arnaud

FONTAINE Maxime CECCOTTI Romain

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

Les génériques sont une classe qui permet à l'utilisateur de définir des classes et méthodes avec un type non défini. L'idée basique derrière l'utilisation des génériques c'est de permettre au type (Integer, String, etc. et les types définis par l'utilisateur) d'être un paramètre des méthodes, classes et interfaces.

Avantages

Une première limite des collections non génériques est l'absence d'une vérification-du-type efficace. Cela signifie que l'on peut mettre n'importe quel objet dans une collection car chaque classe en C# héritent de la classe Objet de Base. Cela compromet la sécurité du type (type safety) et est contradictoire la définition de base du C# en tant que type-safe langage. De plus, utiliser les collections non génériques implique de grosses performances sous la forme d'implicites et explicites type casting, qui est requis pour ajouter ou récupérer un objet de la collection.

Pour répondre à ce problème le framework .NET fournit les génériques pour créer des classes, structures, interfaces et méthodes qui ont un type non défini et un emplacement pour le type qu'elles utilisent. Les génériques sont communément utilisés pour créer justement des collections génériques qui ont donc la propriété du type-safe.

Les génériques en C# sont sa fonction la plus puissante. Le fait que les structures de données soient type-safe permet un remarquable boost de performance et du code de haute qualité, puisque cela permet de réutiliser les algorithmes de process de données sans devoir refaire du code. Les génériques sont similaires aux templates de C++ mais sont différents dans l'implémentation et leurs capacités. Les génériques introduisent le concept de type paramètre, grâce auquel il est possible de créer des méthodes et classes qui reportent la définition du type de données au moment où la classe ou la méthode sont déclarés et sont instanciées par le client code.

Le type générique performe mieux que les systèmes normaux parce qu'ils retirent la nécessite de boxing-unboxing et de type casting pour les variables ou les objets.

Inconvénients

L'inconvénient pourrait être l'ajout de complexité car ajout d'une nouvelle couche d'abstraction de paramétrages en plus de ceux existant. Comme tout puissant outil, il est inutile de l'utiliser pour des choses simples et ce serait une perte de temps, c'est un de ces inconvénients il peut être mal utilisé ou mal compris.

Les génériques dans les collections

La différence basique est que les collections génériques sont fortement typées alors que les non génériques ne le sont pas, sauf si elles ont été spécialement écrites pour n'accepter qu'un seul type de données. Dans le Framework .NET, les collections non génériques (ArrayList, Hashtable, SortedList, Queue, etc.) stockent les éléments de façon interne dans des tableaux d'objet, qui, évidemment, peuvent stocker n'importe quel type de données. Cela signifie que, dans le cas de value-type (int, double, bool, char, etc.) il faut qu'il soit 'boxed' (mis en un objet) puis ensuite 'unboxed' (sorti de l'objet) quand on veut le récupérer de la collection, ce qui est une opération assez lente. Même si cela n'est pas un problème pour les référence-types, il faut quand même les caster en tant que leur vrai type avant de pouvoir les utiliser.

Alors que de l'autre côté, les collections génériques (List<T>, Dictionary<T, U>, Queue<T>, etc.) stockent les éléments de façon interne dans des tableaux de leur vrai type, donc aucun 'boxing', 'unboxing' ou 'cast' ne sera jamais nécessaire. Cela signifie que les collections génériques sont plus rapides que leurs homonymes non génériques lors de l'utilisation de value-type et plus pratiques lors de l'utilisation de référence type. Pour faire court, les non génériques sont maintenant virtuellement superflues.

Créer une classe générique

```
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
```

```
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
```

On définit la classe en précisant que c'est une classe générique grâce aux chevrons et au T qui est une convention pour 'Type'. Puis on lui assigne un attribut value qui est du type T correspondant. On peut ensuite dans le main créer une version 'String', une version 'float', etc. de cette classe et cela fonctionne.

```
Program.cs*
Test
```

```
1   using System;
2
    2 références
3   public class Test
4   {
        3 références
5       public void Afficher<T>(string msg, T value)
6       {
7           Console.WriteLine("{0} : {1}", msg, value);
8       }
9   }
10
    0 références
11  public class Program
12  {
13      // Main Method
        0 références
14      public static int Main()
15      {
16          // on crée un objet de la classe Test.
17          Test p = new Test();
18
19          // On appelle la methode
20
21          p.Afficher<int>("Integer", 122);
22          // Affiche Integer : 122
23
24          p.Afficher<char>("Character", 'H');
25          //Affiche Character : H
26
27          p.Afficher<double>("Decimal", 255.67);
28          //Affiche Decimal : 255.67
29          return 0;
30      }
31  }
```

On créer maintenant une méthode générique du type T qui va afficher un message et une value de type T.

On teste ceci dans le main avec les type double, char, et int, et tout cela fonctionne bien.