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.

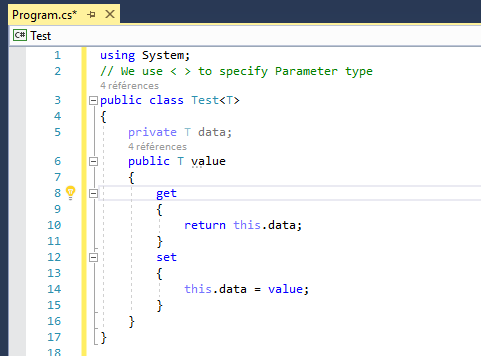
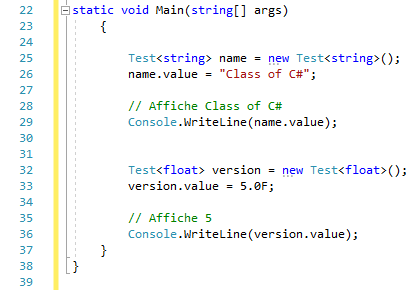
Inconvénients

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.

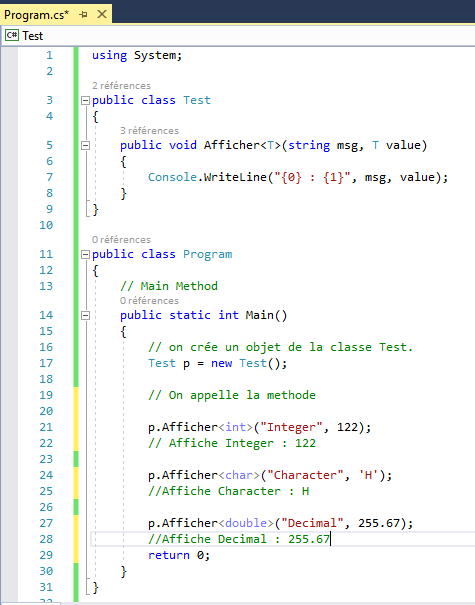
Les génériques dans les collections

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

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.

Créer une méthode générique 

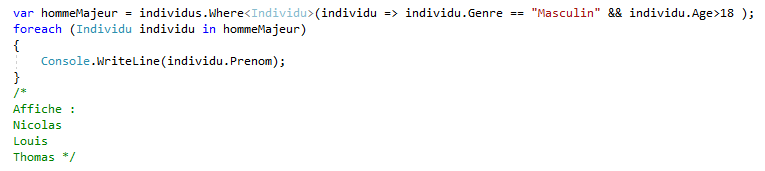
On créer maintenant une méthode générique du type T qui va afficher un message et une value de type T.

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

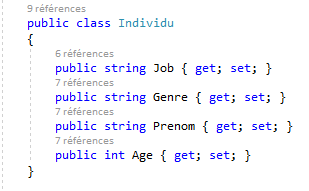
Exercice d’exemple

Pour s’entraîner sur la notion apprise j’ai décidé de proposer un exercice corrigé sur le thème des génériques en créant une méthode générique un peu plus complexe (à priori) qui serait similaire à la méthode Where de LINQ.

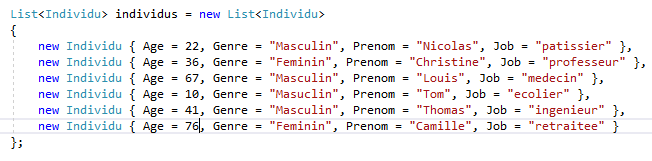
Cette méthode permet de retourner une sous collection de la collection à laquelle on l’applique, qui vérifiera la condition que l’on entre dans un délégate.

En voici un exemple :

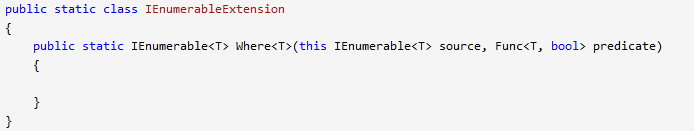
Pour cela il faut d’abord créer une classe :



Puis ensuite créer des instances de cette classe dans une liste :



1ère étape : déclaration de la méthode



Voici la déclaration de la méthode. Je vais expliquer point par point ses différents composants. Même si cela peut à priori paraître compliqué, chaque chose est logique et cela est simple lorsque l’on sait l’explication.  
Il faut d’abord préciser que le <T> (qui est une convention, cela pourrait être <TYPE> ou <U>, etc.) permet d’indiquer la généricité. C’est un PLACE HOLDER, chaque T sera remplacé par le type concerné ensuite. Par exemple, s’il y a <T1> et <T2> alors chaque T1 sera remplacé par le premier type et les T2 par le second.

Voici dans l’ordre :

* Première chose à remarquer : la méthode hérite de « IEnumerable<T> ». C’est de cette interface que toutes les listes génériques de C# héritent. Notre méthode « Where » sera donc une méthode d’extension de cette interface.
* Le type de retour est « IEnumerable<T> ». Ceci est une interface générique. Le <T> permet d’indiquer que cette interface est générique. Cela est nécessaire car notre méthode pourra être appelée sur n’importe quelle collection d’objets et elle pourra gérer n’importe quel type. Et ce sera une collection de ce type qui devras être renvoyé.
* Le nom de la fonction « Where » suivi de tous les types génériques qu’elle va devoir gérer. Ici la méthode ne gérera qu’un seul type, donc on la déclare « Where<T> ». (Si la méthode question devait gérer 2 types par exemple, on aurait pu écrire « Where<T1, T2> »)
* Le premier argument de la méthode sera précédé de this, car le premier argument est la liste à laquelle on appliquera la méthode. Le this permet donc de préciser que l’on peut appliquer la méthode à un objet, qui lui-même sera le premier argument.
* Le premier argument est une collection de n’importe quel type donc de type « IEnumerable<T> ». Comme cela a été expliqué dans le type de retour qui d’ailleurs est du même type que l’objet auquel on l’applique. Et celui-ci s’appellera source car c’est à cette collection source qu’on appliquera notre méthode
* Le deuxième argument est un « delegate » indiqué grâce à « Func » qui va prendre en argument un objet de type <T> (car il va prendre chaque élément de la liste pour les tester) et qui va renvoyer un booléen (pour savoir si l’objet répond ou non à la condition). On écrit donc Func<T, bool>. On l’appellera predicate.
* On remarque que partout dans la fonction il y a toujours le même T pour indiquer la généricité. Il y a une raison pour cela. Cela précise qu’il n’y qu’UN seul et UNIQUE type générique. Cette fonction gère des <T> (exemple : Individu), aura pour premier argument une collection de <T> (Individu), une fonction delegate qui teste des <T> (Individu) et renvoie une collection générique de type <T> (Individu)

2ème étape : contenu de la méthode

Objectif : 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.

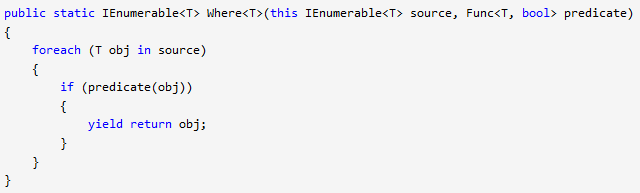
Cela sera très simple à réaliser.

* Pour parcourir les objets on créé une boucle foreach.
* Les objets parcourus seront donc du type T, on écrit donc « T obj ».
* La boucle va parcourir la collection source donc « in source »

On indique maintenant que si notre condition est remplie il faut ajouter l’objet concerné à la liste de renvoi. Pour cela

* On met une condition « if ».
* On applique la fonction delegate à l’objet auquel la boucle foreach est arrivé. On écrit dont « predicate(obj) ».
* Cela nous renvoie directement un booléen indiquant si l’objet répond ou non à la condition et s’il devra se trouver ou non dans la liste de retour. On peut donc directement écrire « if(predicate(obj)) ».
* Maintenant, si l’objet répond à cette condition (donc à cet if) on ajoute l’objet en question dans la liste de retour, pour cela on fera un « yield return obj »

En image :

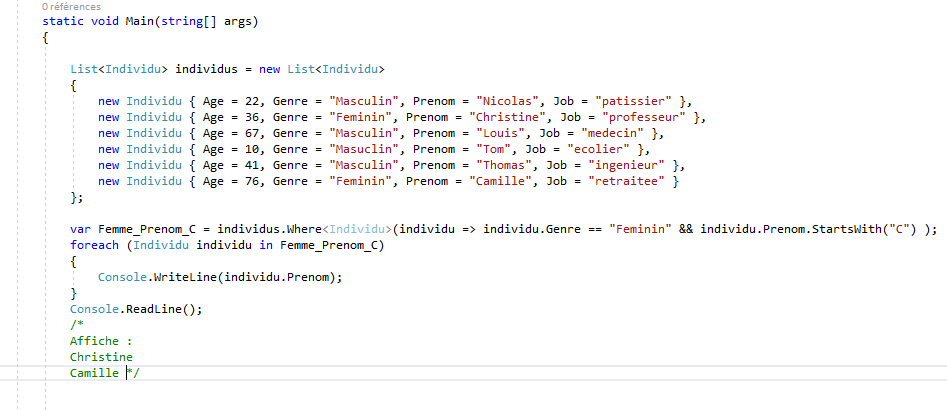


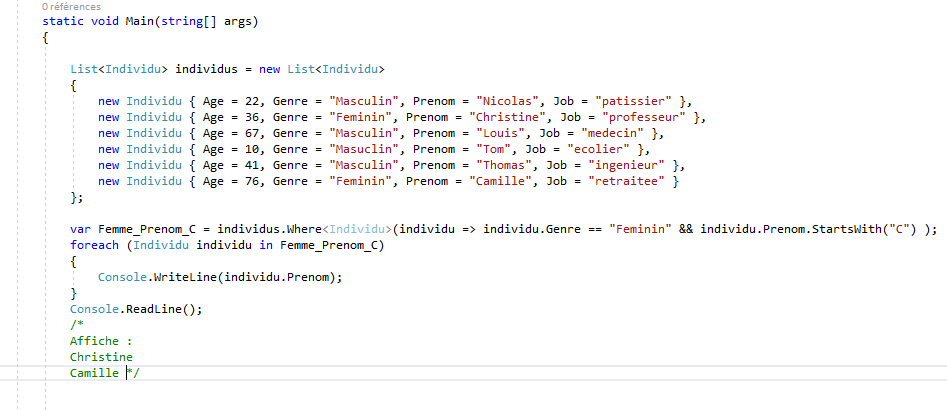
3ème étape : test de la méthode

On va donc rapidement tester la méthode pour vérifier que tout fonctionne comme prévu (et éventuellement débuguer si ça n’était pas le cas).

Pour cela on va l’appliquer à la liste d’individus précédemment créée par nos propres soins. Ainsi la généricité sera bien vérifiée puisque c’est une classe que l’on a créée nous-même et qui n’existe nulle part ailleurs.

On va demander les individus dont le sexe est féminin. Et pour compliquer un peu la tâche on va rajouter que le filtre ne donne que celles dont le prénom commence par la lettre « C ».

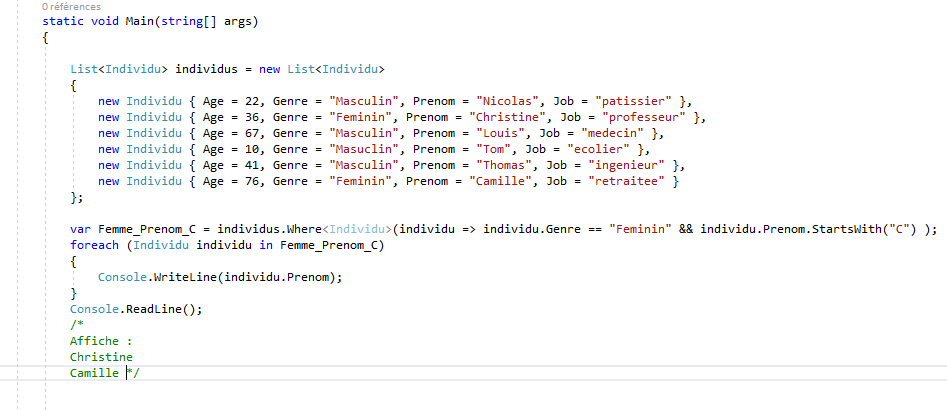
Pour cela on applique notre méthode à la liste en écrivant :

* On indique donc notre objet – point – le nom de la méthode qu’on lui applique. C’est-à-dire « individus.Where ».
* Comme le nom de la méthode n’est pas juste « Where » mais bien « Where<T> » il faut donc rajouter les chevrons et remplacer le T par le type qui nous intéresse. Ici on traite une liste dont chaque objet appartient à la classe « Individu » on doit donc préciser cela dans les chevrons. C’est-à-dire « invididus.Where<Individu> ».
* Puis ouvrir les parenthèses et spécifier les arguments. Le premier est donc la liste à laquelle on applique la méthode, le second sera un delegate. On doit passer un delegate qui correspond à notre condition voulue.  
  Pour cela on écrira :

Cela signifie une fonction qui prend pour argument une variable appelée « individu » et renvoie true si la condition suivante est remplie :  
(le paramètre « genre » de cette variable est égal à « Féminin » ET le paramètre « prénom » de cette variable commence par « C »)

On rajoute la liste retournée à une nouvelle variable « Femme\_Prenom\_C ».

Il ne nous reste plus qu’à la parcourir et afficher les individus pour voir si cela à fonctionner. Rien de plus simple, une boucle foreach qui la parcourt et dont les éléments seront de type « Individu ». Puis un Console.WriteLine des prénoms pour les afficher.



Les individus récupérés sont Christine et Camille, les deux femmes dont le prénom commence par « C ». Tout cela fonctionne comme prévu.

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, qui auparavant été gérée par des « cast » depuis et vers le type universel « Object ». En utilisant les génériques on créer des entités « type-safe » au moment de la compilation qui n’ont pas besoin de « boxing » et « unboxing » non plus ce qui permet de bien meilleures performances pour de grandes listes. Une autre limitation que les génériques ont résout est le manque de vérification de type au moment de la compilation, car avant les génériques puisque tous les type était casté vers « Object » il n’y avait pas de moyen, au moment de la compilation, d’empêcher le client d’ajouter différents types dans la même collection.   
Ils ont de nombreux avantages et peu d’inconvénients et seront indispensables si l’on veut coder quelque chose d’assez développé en C#.