

CHAPITRE 7

TYPES ET MEMBRES GÉNÉRIQUES

7.1 COLLECTIONS GÉNÉRIQUES

Le *framework* .NET offre déjà dans ses bibliothèques un certain nombre de types de collections génériques, notamment dans l'espace de noms `System.Collections.Generic`, comme `ICollection<T>`, `IList<T>` et `IDictionary<TKey, TValue>` qui sont des interfaces implémentées par les classes génériques `List<T>`, `Collection<T>` et `Dictionary<TKey, TValue>`.

Ces types s'instancient comme n'importe quels autres types, sauf qu'il faut dans ce cas déterminer le type `T` des éléments de la collection.

Par exemple :

```
List<int> maCollectionInt = new List<int>();  
List<Joueur> mesJoueurs = new List<Joueur>();
```

Au-delà de l'utilisation conventionnelle de ces collections génériques, nous pouvons nous demander comment nous pouvons nous-mêmes créer de telles collections génériques, mais également comment nous pouvons garder cette genericité lors de l'écriture de méthodes.

7.2 MÉTHODES GÉNÉRIQUES

Il est tout à fait possible de construire une méthode générique, c'est-à-dire une méthode dont la valeur d'un ou plusieurs paramètres est de type qui peut varier. Cela peut être utile :

- Lorsque vous avez plusieurs méthodes surchargées qui effectuent des traitements identiques ou quasi-identiques mais en fonction de types de paramètres différents et qu'il est possible de regrouper toutes ces surcharges vers une seule méthode générique;

- Lorsque le traitement d'une méthode consiste à manipuler des références vers des types ou d'autres éléments et non de faire appel à des fonctionnalités offertes par le type (p. ex. accéder à des métadonnées sur ce type).

Par exemple, imaginons une méthode `Interchanger()` dont l'objectif est d'interchanger les valeurs de deux variables de même type, peu importe le type (p. ex. dans le contexte d'un algorithme de tri sur un tableau d'éléments de type `T` indéterminé).

La méthode générique pourrait alors être la suivante :

```
static void Interchanger<T>(ref T a, ref T b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

Remarquez l'utilisation du mot clé **ref** dans les paramètres. Cela sert à indiquer qu'on est intéressé à la référence vers la variable `a` ou `b`, et non à sa valeur. Comme on ne sait pas si le type `T` sera un type primitif ou un type objet, on souhaite que l'interchange fonctionne peu importe le cas (*on n'est pas du tout obligé d'utiliser `ref` avec les méthodes génériques*)

Pour tester le tout, on pourrait imaginer le code suivant dans la méthode `Main()` :

```
static void Main(string[] args)
{
    int aInt = 3, bInt = 6;
    string aStr = "Adam", bStr = "Joly";

    // interchange des entiers a et b
    Console.WriteLine("-- Interchange d'entiers --");
    Console.WriteLine("[Avant] a: " + aInt + " b: " + bInt);
    Interchanger<int>(ref aInt, ref bInt);
    Console.WriteLine("[Après] a: " + aInt + " b: " + bInt);

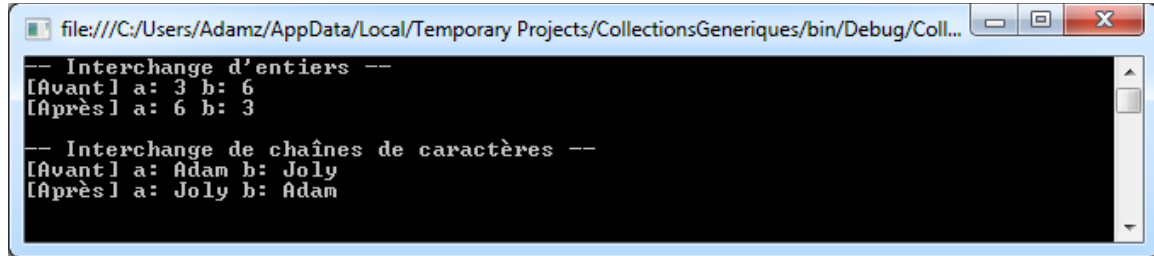
    // interchange des entiers a et b
    Console.WriteLine("\n-- Interchange de chaînes de caractères --");
    Console.WriteLine("[Avant] a: " + aStr + " b: " + bStr);
    Interchanger<string>(ref aStr, ref bStr);
    Console.WriteLine("[Après] a: " + aStr + " b: " + bStr);

    Console.ReadLine();
}
```

Notez que du côté appelant, il faut obligatoirement utiliser le mot clé `ref` pour passer la référence et non la valeur, comme exigé par la méthode générique.

Notez aussi que la spécification du type lorsqu'on appelle la méthode générique peut être omise lorsque ce type peut être inféré.

À l'exécution, on aura donc la sortie suivante :



```
file:///C:/Users/Adamz/AppData/Local/Temporary Projects/CollectionsGeneriques/bin/Debug/Coll...
-- Interchange d'entiers --
[Avant] a: 3 b: 6
[Après] a: 6 b: 3

-- Interchange de chaînes de caractères --
[Avant] a: Adam b: Joly
[Après] a: Joly b: Adam
```

7.3 CLASSES ET STRUCTURES GÉNÉRIQUES

Maintenant, on peut se poser la question comment une classe comme `List<T>` peut être définie à l'interne.

En fait, il est possible de créer des classes ou des structures génériques assez simplement et de leurs ajouter des constructeurs génériques, des variables d'instances génériques, des propriétés génériques et certains mots clés pour nous aider à gérer l'aspect générique de notre construction générique.

Imaginons une structure nommée `Point3D` que nous voulons générique (en ce sens que les types peuvent être des entiers, des décimaux, etc.). La base de notre structure pourrait être la suivante :

```
//-- structure générique--
public struct Point3D<T>
{
    //-- variables d'instances générique--
    private T x;
    private T y;
    private T z;

    //-- constructeur générique --
    public Point3D(T x, T y, T z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    //-- propriétés génériques --
    public T X
    {

```

```
        get { return x; }
        set { x = value; }
    }

    public T Y
    {
        get { return y; }
        set { y = value; }
    }

    public T Z
    {
        get { return z; }
        set { z = value; }
    }

    //-- autres méthodes --
    public void Reinitialiser()
    {
        this.x = default(T);
        this.y = default(T);
        this.z = default(T);
    }

    public override string ToString()
    {
        return string.Format("[{0}, {1}, {2}]", this.x, this.y, this.z);
    }
}
```

La redéfinition de la méthode `ToString()` utilise elle-même la représentation `ToString()` de chacune des variables de type générique.

Par ailleurs, remarquez l'utilisation du mot clé **default**. Vous avez déjà sans doute utilisé ce mot clé à l'intérieur de structures de contrôle `switch`, mais lorsqu'il est utilisé avec des types génériques, il représente la valeur par défaut de ce type :

- Pour un type numérique, la valeur par défaut est de 0;
- Pour des types objets/de référence, la valeur par défaut est `null`;
- Pour une structure, les champs de type numérique sont mis à 0 et ceux de type référence à `null`.

Cela permet de ne pas avoir à « deviner » ce que la valeur par défaut devrait être, ce qui augmente la flexibilité et la simplicité du type générique.

On pourrait donc tester notre structure générique en créant un point de type `int` et un second de type `double`, comme ceci :

```
static void Main(string[] args)
{
    Point3D<int> point = new Point3D<int>(8, 6, 10);
    Point3D<double> pointDbl = new Point3D<double>(4.3, 4.0, 2.2);

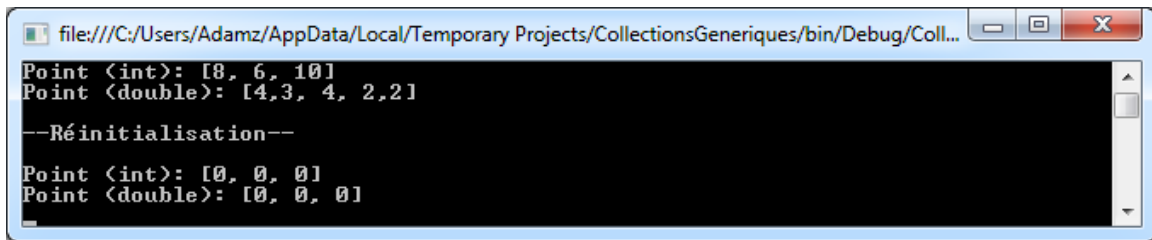
    Console.WriteLine("Point (int): " + point);
    Console.WriteLine("Point (double): " + pointDbl);

    Console.WriteLine("\n--Réinitialisation--\n");
    point.Reinitialiser();
    pointDbl.Reinitialiser();

    Console.WriteLine("Point (int): " + point);
    Console.WriteLine("Point (double): " + pointDbl);

    Console.ReadLine();
}
```

En sortie, on aurait alors :



```
file:///C:/Users/Adamz/AppData/Local/Temporary Projects/CollectionsGeneriques/bin/Debug/Coll...
Point <int>: [8, 6, 10]
Point <double>: [4,3, 4, 2,2]
--Réinitialisation--
Point <int>: [0, 0, 0]
Point <double>: [0, 0, 0]
```

7.4 COLLECTIONS GÉNÉRIQUES PERSONNALISÉES

Maintenant que nous avons créé une structure générique, il est possible de faire la même chose mais dans le but de créer une collection générique.

7.4.1 Syntaxe générale

Évidemment, étant donné le nombre assez diversifié de collections déjà incluses dans le *framework* .NET, créer une classe conteneur générique n'est pas quelque chose qu'on fera tous les jours.

Malgré tout, cela peut être intéressant de montrer comment on peut construire une telle classe.

Imaginons une classe `Robot` toute simple comme ceci :

```
public class Robot
{
    private string nom;
```

```
private int vitesse;

public Robot(string nom, int vitesse)
{
    this.nom = nom;
    this.vitesse = vitesse;
}

public override string ToString()
{
    return this.nom;
}

public string Nom { get { return this.nom; }}
public int Vitesse { get { return this.vitesse; }}
}
```

On pourrait alors créer une classe générique comme précédemment nommé cette fois-ci `RobotCollection<T>` dans laquelle on aura une variable d'instance d'une collection générique `.NET` (ou bien un tableau bien normal, moyennant beaucoup plus de code pour contrôler les fonctionnalités de gestion et manipulation de la collection).

On souhaitera aussi dans ce cas faire implémenter à notre classe conteneur générique l'interface `IEnumerable<T>`, une interface générique qui hérite elle-même de `IEnumerable`, qui permettra à `RobotCollection<T>` de supporter les parcours itératifs de la collection à l'aide de la structure de contrôle `foreach`, ce qui est fortement souhaitable pour toute collection.

On inclura à titre d'exemple quelques fonctionnalités de base de toute collection, comme récupérer un élément situé à un index particulier dans la collection (`Robot()`) ajouter un élément (`Add()`), effacer les éléments de la collection (`Clear()`) et (comme propriété) compter le nombre d'éléments dans la collection (`Count`). Le résultat pourrait être ceci :

```
//-- la collection générique implémente IEnumerable<T> pour permettre
l'utilisation du foreach --
public class RobotCollection<T> : IEnumerable<T>
{
    private List<T> robots = new List<T>(); // on réutilise le type g
    énérique pour créer une liste générique du même type générique

    //-- on ajoute des méthodes avec des paramètres génériques --
    public T Robot(int index)
    {
        return robots[index];
    }

    public void Add(T robot)
    {
        robots.Add(robot);
    }
}
```

```
public void Clear()
{
    robots.Clear();
}

public int Count { get { return robots.Count; } }

//-- il faut implémenter les méthodes GetEnumerator() de l'interface --
//    IEnumerable<T> ET IEnumerable (dont IEnumerable<T> hérite)
IEnumerator<T> IEnumerable<T>.GetEnumerator()
{
    return robots.GetEnumerator();
}

System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator()
{
    return robots.GetEnumerator();
}
}
```

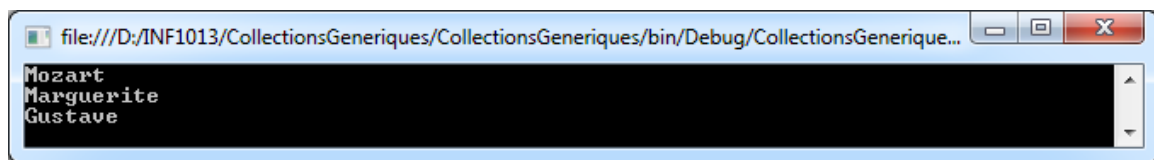
On pourrait alors utiliser cette collection générique de l'extérieur comme ceci :

```
class Program
{
    static void Main(string[] args)
    {
        RobotCollection<Robot> robots = new RobotCollection<Robot>();
        robots.Add(new Robot("Mozart", 10));
        robots.Add(new Robot("Marguerite", 8));
        robots.Add(new Robot("Gustave", 12));

        foreach (Robot robot in robots)
            Console.WriteLine(robot);

        Console.ReadLine();
    }
}
```

Ce qui donnerait en sortie :



Évidemment, on pourrait utiliser directement la collection générique `List<Robot>` pour effectuer ces opérations, mais l'avantage de la collection personnalisée pourrait être d'ajouter des méthodes supplémentaires personnalisées liées à la manipulation des éléments, par exemple.

En fait, un des problèmes d'une collection générique telle que définie plus haut est qu'elle ne permet pas réellement de pouvoir ajouter des méthodes très particulières étant donné qu'on ne connaît pas d'avance de quel type seront les éléments de la collection (cela va de soi, sinon on n'aurait pas une collection dite générique!!!). Ainsi, rien ne nous empêche dans ce cas de ne pas ajouter des robots, mais plutôt des entiers (`int`) ou même des points.

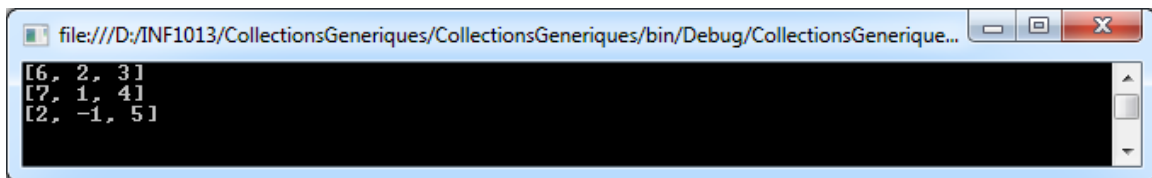
Par exemple :

```
static void Main(string[] args)
{
    RobotCollection<Point3D<int>> robots = new
    RobotCollection<Point3D<int>>();
    robots.Add(new Point3D<int>(6, 2, 3));
    robots.Add(new Point3D<int>(7, 1, 4));
    robots.Add(new Point3D<int>(2, -1, 5));

    foreach (Point3D<int> robot in robots)
        Console.WriteLine(robot);

    Console.ReadLine();
}
```

En sortie :



Cela fonctionne syntaxiquement à cause de la généricité de la collection, mais sémantiquement, ça ne va évidemment pas! On pourrait souhaiter garder l'aspect générique tout en ne permettant qu'à certains types de pouvoir être admis (donc tracer certaines limites autour de la généricité).

Dans cette optique, on pourrait imaginer une classe dérivée de robot comme un robot spécialisé tel que `RobotAspirateur` qui fournit des services supplémentaires (p. ex. `Aspirer()`):

```
public class RobotAspirateur : Robot
{
    public RobotAspirateur(string nom, int vitesse)
        : base(nom, vitesse){}

    public void Aspirer()
    {
        Console.WriteLine("Woouooooooooooooooooouush!");
    }
}
```



```
    }
}
```

Déjà, présentement, rien ne nous empêche à priori d'ajouter et des robots réguliers et des robots aspirateurs dans une collection de type `Robot` :

```
static void Main(string[] args)
{
    RobotCollection<Robot> robots = new RobotCollection<Robot>();
    robots.Add(new Robot("Mozart", 10));
    robots.Add(new RobotAspirateur("Marguerite", 8));

    foreach (Robot robot in robots)
        Console.WriteLine(robot);

    Console.ReadLine();
}
```

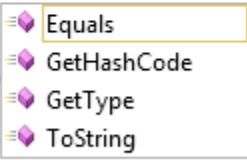
Disons maintenant qu'on souhaiterait ajouter une méthode à la classe conteneur générique `RobotCollection` expressément introduite pour traiter des données en lien avec des robots ou des types dérivés. Par exemple, considérons une nouvelle méthode `GetNomRobot()` :

```
public void Clear()
{
    robots.Clear();
}

public Robot GetNomRobot(int index)
{
    Console.WriteLine(robots[index].|
}

public int Count { get { return robo

//-- il faut implémenter les méthode
// IEnumerable<T> ET IEnumerable (dont IEnumerable<T>
IEnumerator<T> IEnumerable<T>.GetEnumerator()
}
```



Si on tente d'accéder à un des éléments de la collection, étant donné que la collection `robots` n'est pas d'un type spécifique, on n'a donc pas accès aux propriétés et méthodes de la classe `Robot`.

Même si on voulait « tricher » quelque peu en utilisant une opération de conversion, le compilateur ne nous laisserait pas faire, étant donné qu'il ne peut pas garantir que la conversion serait sécuritaire car il n'est pas assuré que le type choisi sera `Robot` :

```

public Robot GetNomRobot(int index)
{
    Console.WriteLine(((Robot)robots[index]).Nom);
}

public int Count { get
//-- il faut implémenter

```

List<T> RobotCollection<T>.robots

Error:
Cannot convert type 'T' to 'CollectionsGeneriques.Robot'

Que faire, et à quoi sert dans ce cas les collections génériques personnalisées?

7.4.2 Where : contraintes sur des collections génériques personnalisées

Une bonne nouvelle maintenant : .NET permet de doter un paramètre de type générique d'une contrainte qui délimite à certains égards les types admissibles, grâce au mot clé **where** et à une syntaxe particulière.

C'est d'ailleurs là tout l'intérêt de construire des collections génériques.

Voici un tableau qui résume les différentes façons de contrôler les types admissibles :

Contrainte générique	Description de la contrainte
where T : struct	Le paramètre de type <T> doit être une structure (et donc hérité de <code>System.ValueType</code>)
where T : class	Le paramètre de type <T> ne doit pas hériter de <code>System.ValueType</code> , c'est-à-dire qu'il doit être un type de référence (une classe)
where T : new()	Le paramètre de type <T> doit absolument posséder un constructeur par défaut
where T : <i>NomClasseBase</i>	Le type <T> doit dériver de la classe spécifiée
where T : <i>NomInterface</i>	Le type <T> doit dériver de l'interface spécifiée

Notez que plusieurs contraintes peuvent (et parfois doivent) être employées à la fois. Vous êtes invités à faire différents tests avec un ou plusieurs paramètres de type générique.

Dans ce chapitre, nous nous contenterons d'utiliser une contrainte permettant de limiter à une classe et ses dérivées les types possibles pour une collection générique.

Reprenant notre exemple précédent, on pourrait alors modifier la classe `RobotCollection<T>` de façon à indiquer qu'on souhaite que notre collection ne comporte que des robots, peu importe le type de robot :

```
public class RobotCollection<T> : IEnumerable<T> where T : Robot
{
    private List<T> robots = new List<T>(); // on réutilise le type
    générique pour créer une liste générique du même type générique

    (...)

    public void GetNomRobot(int index)
    {
        Console.WriteLine(robots[index].Nom); // il n'y a plus de problème
        maintenant ici
    }

    (...)
}
```

On peut alors faire appel à des méthodes, des propriétés ou d'autres membres de nos éléments de façon sécuritaire, étant assuré qu'ils seront de type `Robot` (ou des types dérivés).

Du côté appelant, on ne sera plus en mesure d'ajouter des éléments de tout type (une erreur sera indiquée) :

```
static void Main(string[] args)
{
    RobotCollection<Robot> robots = new RobotCollection<Robot>();
    robots.Add(new Robot("Mozart", 10));
    robots.Add(new RobotAspirateur("Marguerite", 8));
    robots.Add(10);
}

void RobotCollection<Robot>.Add(Robot robot)

Error:
The best overloaded method match for 'CollectionsGeneriques.RobotCollection<CollectionsGeneriques.Robot>.Add(CollectionsGeneriques.Robot)' has some invalid arguments
```

7.5 CLASSES DE BASE GÉNÉRIQUES

Par ailleurs, il est possible d'utiliser des classes génériques en tant que classes de base pour d'autres classes, pour lesquelles on détermine alors le type du paramètre.

Par exemple, réutilisons la structure générique `Point3D<T>`. Faisons-en une classe abstraite pour les besoins de la cause :

```
public abstract class Point3D<T>
{
```

```
//-- variables d'instances générique--
protected T x;
protected T y;
protected T z;

//-- constructeur générique --
public Point3D(T x, T y, T z)
{
    this.x = x;
    this.y = y;
    this.z = z;
}

(...)

//-- autres méthodes --
public void Reinitialiser()
{
    this.x = default(T);
    this.y = default(T);
    this.z = default(T);
}

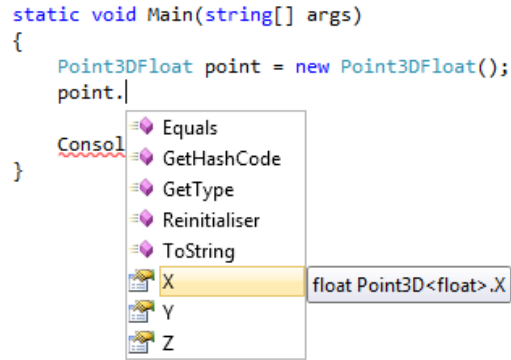
public override string ToString()
{
    return string.Format("[{0}, {1}, {2}]", this.x, this.y, this.z);
}
}
```

Nous pouvons ensuite créer une classe dérivée de cette dernière dans laquelle nous spécifions, par exemple qu'il s'agira d'un point ayant des valeurs de type `float` :

```
public class Point3DFloat : Point3D<float>
{
}
```

Ainsi, il suffit de déclarer une variable de type `Point3DFloat` pour créer un type `Point3D` dont le type spécifique du paramètre est un `float`.

On a alors accès à toutes les méthodes et propriétés de cette classe :



Comme vous pouvez le constater, ce faisant, pour les méthodes héritées où on utilisait le type générique `T` soit dans les paramètres, soit dans les valeurs de retour, soit lors du traitement, `T` est remplacé partout par le type défini (`float` dans ce cas).

Plus encore, on pourrait se servir de la classe de base générique pour définir des méthodes virtuelles ou abstraites que la classe dérivée spécifique devra nécessairement redéfinir.

Par exemple, imaginons qu'on ajoute cette méthode à la classe abstraite générique :

```
public abstract void Arrondir();
```

Il faudra ensuite définir dans toutes les classes implémentant cette classe générique comment l'opération d'arrondir est effectuée. Par exemple, pour un `float`, on pourrait considérer un arrondissement à 3 décimales :

```

public class Point3DFloat : Point3D<float>
{
    public Point3DFloat(float x, float y, float z) : base(x, y, z){}

    public override void Arrondir()
    {
        this.x = (float) Math.Round(this.x, 3);
        this.y = (float) Math.Round(this.y, 3);
        this.z = (float) Math.Round(this.z, 3);
    }
}

```

La méthode `Main` de test :

```

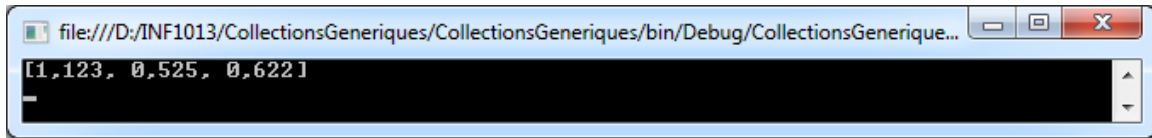
static void Main(string[] args)
{
    Point3DFloat point = new Point3DFloat(1.1235f, 0.5253f, 0.6224f);
    point.Arrondir();

    Console.WriteLine(point);
}

```

```
        Console.ReadLine();  
    }
```

En sortie :



Le même principe s'applique également aux interfaces.