TPC#12 : Génériques, interfaces et délégués !

Table des matières

1 Structure du rendu

La structure du rendu pour ce TP ne change pas de celles que vous avez l'habitude de voir. Encore une fois, `login_x` est à remplacer par votre propre login.

```
rendu-tpcs12-login_x.zip
    - login_x/
        - AUTHORS
        - Interface
              - Interface (tout sauf bin et obj)
              - Interface.sln
        - MyGenericTab
              - MyGenericTab
              - MyGenericTab.sln
```

2 La généricité

2.1 Principe de la généricité

Les types génériques permettent de factoriser une partie de votre code en décrivant des classes ou des méthodes dont la spécification d'un ou de plusieurs types n'est pas donnée jusqu'à ce que la classe ou la méthode soit déclarée et instanciée. Leur usage permet de gagner en performance et de sécuriser les types lors de la compilation (les génériques évitent d'utiliser les casts, parfois dangereux).

L'utilisation la plus courante des classes génériques est avec des collections comme les listes triées, les tables de hachage, les piles, les files d'attente, les arborescences et autres collections où les opérations telles que l'ajout et la suppression d'éléments de la collection sont exécutées de façon relativement similaire indépendamment du type des données qui sont stockées. Voici un exemple de type que vous utilisez depuis longtemps utilisant les génériques, le type List :

```
// Déclaration d'une liste contenant des entiers
List<Int> my_list = new List<int>();
// Déclaration d'une liste contenant des ACDC
List<ACDC> DreamTeam = new List<ACDC>();
```

On voit bien qu'on peut créer des listes de n'importe quel type avec ce procédé.

Plus concrètement, la généricité peut s'appliquer à :

- une classe
- une structure
- une méthode
- une interface
- un délégué

Elle permet de dire que votre ... truc générique peut prendre en compte un type quelconque, que ce soit de valeur ou de référence. Il est dès lors possible de factoriser des lignes de codes, et de s'éviter des efforts inutiles.

2.2 Créer des classes génériques

Aujourd'hui, nous allons aller de l'autre côté de la template en apprenant à faire des classes génériques ! Pour décrire une classe générique, il suffit d'expliquer qu'elle dépend d'un type non fixé.
Voici un petit exemple commenté pour expliquer comment procéder :

```
// Dépend d'un type inconnu qu'on nomme T
public class MyBox<T>
{
      // Utilisation de T comme un type habituel
      public T value;
      // Prend un paramètre de type T
      public MyBox(T val) { value = val; }
      // Renvoie une valeur de type T
      public T getValue() { return value; }
}

static void Main()
{
      MyBox<int> first_box = new MyBox<int>(2);
      MyBox<string> second_box = new MyBox<string>("two");
      Console.WriteLine(first_box.getValue()); // affiche "2"
      Console.WriteLine(second_box.getValue()); // affiche "two"
}
```

Super facile non ?

2.3 Contraintes

Lorsque vous définissez une classe générique, vous pouvez appliquer des restrictions aux genres de types que l'utilisateur peut utiliser pour les arguments de type lorsqu'il instancie votre classe. Si l'utilisateur essaie d'instancier votre classe avec un type qui n'est pas autorisé par une contrainte, il en résultera une erreur de compilation. Ces restrictions sont appelées des **contraintes**. Les contraintes sont spécifiées à l'aide du mot clé contextuel **where**.

Il existe plusieurs types de contraintes, en voici quelques-unes :

- classe : L'argument de type doit être un type référence, y compris tout type classe, interface, délégué ou tableau.
- <nom d'interface> : L'argument de type doit être ou doit implémenter l'interface spécifiée. Plusieurs contraintes d'interface peuvent être spécifiées. L'interface qui impose les contraintes peut également être générique.
- <nom de classe de base> : L'argument de type doit être ou doit dériver [1] de la classe de base spécifiée.

Il y en a d'autres [2], on vous a listé les plus importantes.

Voici un exemple de contrainte :

```
class G<Y,Z> where Y : IComparable
```

Ici, le type Y doit respecter l'interface [3] IComparable. Il peut aussi y avoir plusieurs contraintes pour un type : elles sont séparées par des virgules.

3 Interfaces

3.1 Principe des interfaces

Les interfaces sont un concept très important de la programmation orientée objet. Elle consiste à définir un ensemble de fonctionnalités comme des méthodes, des propriétés, des évènements et des indexeurs. Les classes ou les structures qui implémentent une interface doivent impérativement implémenter toutes les méthodes et propriétés définies dans l'interface.

On utilise le mot-clef "interface" pour les déclarer. Il n'y a aucun corps de fonction et on ne peut instancier une interface malgré sa proximité avec les classes. Une classe et une structure peuvent implémenter plusieurs interfaces à la fois.

Une interface ne peut pas contenir de constantes, de champs, d'opérateurs, de constructeurs d'instances, de destructeurs ou de types. Elle ne peut pas contenir de membres statiques. Les membres d'interface sont automatiquement publics et ils ne peuvent contenir aucun modificateur d'accès.

3.2 IEnumerable

Le C# propose pleins d'interfaces dont la majorité utilise les génériques. N'hésitez pas à vous renseigner dessus. En attendant, voici un exemple d'interface de C# sexy : IEnumerable est une interface permettant de parcourir un conteneur. Celle-ci permet entre autres d'utiliser un objet dans une boucle "foreach". La seule méthode à implémenter pou respecter cette interface est :

```
IEnumerator GetEnumerator()
```

3.3 Implémenter sa propre interface

Rien ne vous empêche de définir votre propre interface. Par convention, leur nom commence par un "I" et les deux premières lettres sont en majuscules. Exemple :

1. Héritage, toussa toussa...
2. Voir "Plus en profondeur..."
3. Voir la partie "Interfaces"

```
interface IDimensions
{
      float getLength();
      float getWidth();
}

class Box : IDimensions
{
      float lengthInches;
      float widthInches;

      Box(float length, float width)
      {
            lengthInches = length;
            widthInches = width;
      }

       // Implémentation implicite mais obligatoire de IDimensions
      float getLength() { return lengthInches; }
      float getWidth() { return widthInches; }

      static void Main()
      {
           Box box1 = new Box(30.0f, 20.0f);

           // Affiche les dimensions de la boite grâce aux méthodes depuis l'objet
           Console.WriteLine("Length : {0} ", box1.getLength());
           // affiche "Length : 30 "
           Console.WriteLine("Width : {0} ", box1.getWidth());
           // affiche "Width : 20 "
      }
}
```

Voici le même exemple mais avec une implémentation explicite des interfaces :

```
interface IDimensions
{
      float getLength();
      float getWidth();
}

class Box : IDimensions
{
      float lengthInches;
      float widthInches;

      Box(float length, float width)
      {
            lengthInches = length;
            widthInches = width;
      }

       // Implémentation explicite mais obligatoire de IDimensions
      float IDimensions.getLength() { return lengthInches; }
      float IDimensions.getWidth() { return widthInches; }

      static void Main()
      {
           Box box1 = new Box(30.0f, 20.0f);
           IDimensions dimensions = (IDimensions)box1;
           Console.WriteLine("Length {0}", dimensions.getLength());
           Console.WriteLine("Width {0}", dimensions.getWidth()):
      }
}
```

3.4 Exercice

Créer le projet Interface dans le dossier Interface.

- Créez une interface ISport avec au moins quatre fonctionnalités de votre choix.
- Créez les classes Volleyball, Football et Rugby de manière à ce qu'elle implémente ISport. Le reste est libre !
- Dans votre Main(), faites des tests avec des instances des trois classes.

4 Les délégués

Les **délégués** (*delegate* en anglais) est un type qui définit une signature de méthode (et son type de retour). Quand vous instanciez un délégué vous associez l'instance de votre délégué (la variable pour ceux qui seraient un peu perdus) avec une méthode dont la signature est la même que celle définie dans le délégué.

Voici à quoi ça ressemble :

```
public delegate int MyDelegate(int x);
class Program
{

    static int square(int x)
    {
        return x * x;
    }

    static int minusFortyTwo(int x)
    {
        return x - 42;
    }

    static void Main(string[] args)
    {
        int x = 12;

        MyDelegate intFunctions = square;
        Console.WriteLine(intFunctions(x));

        intFunctions = minusFortyTwo;
        Console.WriteLine(intFunctions(x));
    }
}
```

Recopiez ce code, regardez ce qu'il fait et essayez de comprendre pourquoi.

Il faut tout d'abord remarquer une chose : il y a deux fonctions, square et minusFortyTwo. Ces deux fonctions ont la même signature et le même type de retour. Elle ne font pas la même chose mais elles ont le même nombre et types d'arguments et le même type de retour.

A la première ligne on crée un nouveau délégué appelé MyDelegate. Ensuite dans la fonction main on instancie ce délégué, en créant la variable *intFunctions*. On lui affecte la méthode *square* (sans parenthèses, on est pas en train de faire un appel de fonction). Ensuite on appelle notre délégué exactement comme une fonction.

```
intFunctions(x)
```

Ceci appelle la méthode [4] qui est dans le délégué *intFunctions*. Une fois votre délégué instancié il peut s'utiliser exactement comme une fonction.

Pour mieux comprendre ce qu'est un délégué entrainez vous, faites vos propres essais.

Quel est l'utilité ?

Voici un exemple. Vous créez une classe pour représenter votre structure de données préférée (liste, arbre, tas de Fibonacci [5]), avec notamment une fonction de tri dans cette structure. Mais vous êtes super forts, du coup vous avez fait plusieurs algorithmes pour trier vos données. Quand vous appelez la fonction de tri vous voulez pouvoir dire "je veux que tel algorithme soit appelé".

Vous pourriez faire :

4. Une instance de délégué peut contenir plusieurs méthodes. En effet vous pouvez écrire *intFunctions = minusFortyTwo ;* puis *intFunctions += square ;* Je vous laisse voir ce que ça fait par vous même.
5. celui qui implémente un tas de Fibonnaci pendant le TP aura beaucoup de points bonus

```
enum SortAlgo { Bubble, Quick, Intro, Heap };
public void mySort(SortAlgo algo)
{
        //do some stuff
        if (algo == SortAlgo.Bubble)
        //call bublesort
        else if (algo == SortAlgo.Heap)
        //call heap sort
        else if (algo == SortAlgo.Intro)
        //call intro sort
        else
        //throw exception
        //do some other stuff
}
```

Et dès que vous ajoutez un algorithme de tri, il faut modifier l'enum et rajouter un if.
Avec des délégués vous pouvez faire :

```
public delegate void SortAlgo(MySuperDataStruct c);
public void mySort(SortAlgo algo)
{
           //do some stuff
            algo(/* data to sort*/);
            //do some other stuff
}
```

Voilà comment remplacer tous ces if pas beaux et longs (chiants ?) à écrire. Bien sûr vous pouvez trouver beaucoup d'autres utilisations aux delegates, parfois ils ne seront pas nécessaires mais leur utilisation vous facilitera la vie.

5 Fonctions

5.1 Fonctions variadiques

Vous savez depuis longtemps comment créer des fonctions avec arguments. Mais savez-vous créer des fonctions avec un nombre variable d'arguments ?
Prenons un exemple, vous avez une fonction qui va écrire dans un fichier (avec un Stream) différents objets. Pour cela, vous pourriez créer une fonction de ce genre :

```
void WriteInFile(String path, object l)
{
    Stream stream = new FileStream(path, FileMode.Append, FileAccess.Write);
    StreamWriter sw = new StreamWriter(stream);

    sw.Write(l);
    sw.Flush();

    sw.Close();
    stream.Close();
}

static void Main(string[] args)
{
 WriteInFile("file", 5);
 WriteInFile("file", 42);
 WriteInFile("file", "Object");
 WriteInFile("file", 56);
}
```

Comme vous pouvez le voir, tout ça n'est pas très pratique si on veut effectuer des écritures à la suite. Bien entendu, on pourrait changer le "object l" par un tableau d'Object ("object[] l"), puis faire un parcours, tout ça nous obligerait à créer un tableau avant de faire l'appel à la fonction WriteInFile.

C'est là que le terme "variadique" entre en scène. Le principe des fonctions variadiques est simple : permettre l'appel à une fonction avec un nombre variable d'objets. Pour ceux qui ont déjà codé en C, rappelez-vous la fonction "printf".
Pour réaliser ceci, nous avons besoin d'un nouveau mot clé : **params**. Voici la définition et l'exemple fournis par MSDN :

> The params keyword lets you specify a method parameter that takes an argument where the number of arguments is variable.
> No additional parameters are permitted after the params keyword in a method declaration, and only one params keyword is permitted in a method declaration.

```
public static void UseParams(params int[] list)
{
    for (int i = 0; i < list.Length; i++)
        Console.WriteLine(list[i]);
    Console.WriteLine();
}

public static void UseParams2(params object[] list)
{
    for (int i = 0; i < list.Length; i++)
        Console.WriteLine(list[i].GetType() + " " + list[i]);
    Console.WriteLine();
}

static void Main(string[] args)
{
    UseParams(1, 2, 3);
    UseParams2(1, 'a', "test");

    int[] myarray = new int[3] {10,11,12};
    UseParams(myarray);
}
```

Comment vous pouvez le voir, le mot params s'utilise de cette manière "(params typename[] name)". Vous pouvez appeler la fonction avec soit un tableau du type "typename", soit avec plusieurs élements du type "typename" séparés par une virgule.

Dans UseParams2, vous pouvez remarquer l'utilisation du mot clé **object**, qui peut en fait représenter tous les types possibles.
Il représente en fait un alias (un raccourci, si vous préférez) pour la classe Object. Cette classe est en fait la mère de toute chose, le père de vos pères, la mère du fils de votre soeur, et même celle de votre feu grand oncle Marcel.
Plus sérieusement, maintenant que vous êtes plus ou moins familier avec la notion d'héritage, nous pouvons vous dire la vérité : toutes les classes héritent directement ou indirectement de cette classe. Vous trouverez plus d'informations sur MSDN ou auprès des vos ACDCs (encore une fois). C'est donc pour cette raison que vous pouvez placer le mot object pour représenter tous les types : ils héritent tous de lui. Dans le premier exemple, où l'on écrivait " Stream stream = new FileStream(...) ", le compilateur ne sortait pas d'erreurs car FileStream hérite de Stream, et donc la conversion de FileStream vers un Stream ne posent aucun problème (notez que l'inverse n'est pas vrai, nous vous laissons deviner pourquoi).

Maintenant que vous savez tout ceci, essayez d'appeler UseParams2 avec comme paramètre "new int[3]1, 2, 3". Nous vous laissons deviner tout seul pourquoi le résultat n'est pas celui que vous attendiez.

5.2 Fonctions anonymes

Nous allons voir ce que sont les fonctions anonymes. Comme vous pouvez vous en douter, une fonction anonyme est une fonction "sans nom". Ça ne vous aide pas beaucoup, et puis une fonction sans nom c'est un peu con, non ? Comment l'appeler si elle n'a pas de nom ?

Allons voir ce que MSDN nous dit là dessus.

> Une fonction anonyme est une instruction ou expression "inline" qui peut être utilisée partout où un type délégué est attendu. Vous pouvez l'utiliser pour initialiser un délégué nommé ou la passer à la place d'un type délégué nommé en tant que paramètre de méthode. MSDN

Retenez donc qu'une fonction anonyme peut être utilisée partout où un délégué est attendu.

Entrons plus dans le détail pour voir deux types de fonctions anonymes : les expressions lambda et les méthodes anonymes (notez la différence entre *fonctions anonymes* et *méthodes anonymes*).

5.2.1 Méthodes anonymes

Les méthodes anonymes servent entre autres à passer un bloc de code en paramètre de délégué, c'est à dire que vous pouvez créer la méthode qui est encapsulée par votre délégué directement dans l'instanciation de celui-ci.

Voici un exemple :

```
// Création d'un délégué
delegate void Del(int x);

// Instanciation du délégué avec une méthode anonyme
Del d = delegate(int k) { /* votre code ici */ };
```

Vous pouvez écrire dans votre méthode anonyme (là où il y a *votre code ici*) comme si vous codiez dans une fonction prenant un int en paramètre et ne renvoyant rien. Comme vous pouvez le voir la syntaxe est assez simple.

Vous ne voyez peut-être pas l'utilité pour l'instant, mais une fois que vous serez habitués aux délégués vous serez contents de pouvoir utiliser les méthodes (et fonctions en général) anonymes au lieu d'en créer une explicite. C'est plus rapide à faire et parfois plus explicite.

Attention, il ne faut pas en abuser et utiliser les méthodes anonymes partout, ne remplacez pas toutes vos méthodes par des méthodes anonymes (en plus nous ne pensons pas que vous ayez envie de le faire). Utilisez les à bon escient.

5.2.2 Expressions lambda

Les **expressions lambda** (*lambda expressions* en anglais, oui oui c'est une remarque utile). Encore une fois, une explication venant de MSDN (MSDN est votre ami).

> A lambda expression is an anonymous function that you can use to create delegates or expression tree types. By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls. Lambda expressions are particularly helpful for writing LINQ query expressions.
>
> To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator =>, and you put the expression or statement block on the other side. For example, the lambda expression x => x * x specifies a parameter that's named x and returns the value of x squared.

Si vous ne comprenez pas la partie sur les types d'arborescence ou sur LINQ c'est normal, et ne vous inquiétez pas nous n'utiliserons pas ces notions pendant le TP.

La syntaxe encore une fois est assez simple[6] :

```
(input parameters) => expression
```

6. Ça devrait vous rappeler le mot clef *function* en Caml

Si vous n'avez qu'un seul paramètre, alors les parenthèses sont facultatives. Le type des paramètres est lui aussi facultatif dans certains cas (quand le compilateur n'arrive pas à trouver le type des paramètres tout seul).

```
() => 2 // aucun paramètre
(x, y) => x == y // pas besoin des types de x et y
(int x, string s) => s.Length > x  // il faut spécifier les types
```

6 Exercice

6.1 Tableau dynamique générique et délégués

Dans cette partie, on va implémenter notre propre structure de données : un tableau dynamique. Dynamique signifie que sa taille peut changer. Si on arrive à la taille max du tableau, on le réalloue avec une taille plus grande tout en récupérant ses données.

Etant donné que vous avez appris de nouvelles notions lors de la lecture de ce TP, on va les mettre en application : on va implémenter un tableau dynamique générique, tout en se servant d'interface, de délégués et de fonctions anonymes.

On vous donnera les méthodes à implémenter ainsi que leur signature et ce qu'elles doivent faire, et quelle "notion" utiliser... à vous de faire le reste !

Notre classe, se nommera MyGenericTab. Attention ! Elle doit être générique, et on vous **impose** une contrainte sur le type : il doit hérité de l'interface IComparable.

Elle aura comme attributs :
- Un tableau de type T non initialisé
- Un entier contenant le nombre d'éléments actuels dans le tableau

Les méthodes :
- Un constructeur : initialise le tableau ainsi que son nombre d'éléments contenus
```
public MyGenericTab<T>() { ... }
```
- Insert : ajoute un élément à la première position libre du tableau, si le tableau est plein, on le réalloue en doublant la taille du tableau !
```
public void Insert(T elt) { ... }
```
- Delete : supprime l'élément passé en argument dans le tableau, et on redécale tous les éléments pour qu'il n'y ait pas de trous. Renvoie vrai si la suppression a marché, faux sinon.
```
public bool Delete(T elt) { ... }
```
- At : renvoie la ième valeur. Déclenche une exception si l'index est invalide.
```
public T At(int i) { ... }
```
- mapDelegate : délégué à utiliser pour map. Il est à déclarer à l'extérieur de la classe.
```
public delegate T mapDelegate<T>(T elt);
```
- Map : Applique le délégué passé en paramètre à tous les éléments du tableau
```
public void Map(mapDelegate funct) { ... }
```
- Sort : Trie le tableau en se servant de la méthode CompareTo de IComparable
```
public void Quicksort(int left, int right) { ... }
```
- Print : Affiche en vert les éléments pour lesquels le prédicat est passé en paramètre (délégué), sinon affiche l'élément en rouge.
```
public void Print(/* délégué qui renvoit un booléen et prend en paramètre
                                  un élément T */) { ... }
```

6.2 Bonii

- Insert amélioré : l'insertion se fait en respectant l'ordre du tableau
- Delete amélioré : Si seulement un tiers du tableau est utilisé, on réalloue le tableau en divisant sa taille par 2.
- Select (cherchez ce que c'est)

7 Plus en profondeur...

Cette partie est là pour ceux qui désirent en savoir plus, car ce qui a été montré n'englobe pas toutes les fonctionnalités des notions citées en C#.

On vous montre juste quelques particularités qui existent, si vous voulez en savoir plus, google est votre ami, ou profitez-en pour essayer de mettre une colle à votre ACDC.

- Type valeur
- Type référence
- Mot clé default pour la généricité
- Typage sécurisé lors de la compilation grâce à la généricité
- Contrainte new() sur un paramètre générique
- On peut implémenter un membre d'interface explicitement : void ISampleInterface.SampleMethod(), mais c'est pas aussi simple...