TPC#12 : Generics, interfaces and delegates !

Contents

1 Submit architecture

The architecture for the submit of this practical work does not change much from the ones you are used seeing. Once again, `login_x` is to be replaced by your own login.

```
rendu-tpcs12-login_x.zip
    - login_x/
        - AUTHORS
        - Interface
              - Interface (everything except bin and obj directories)
              - Interface.sln
        - MyGenericTab
              - MyGenericTab
              - MyGenericTab.sln
```

2 Generics

2.1 Principle of generics

Generics allow the user to factorize a part of his code by describing classes or methods whose specificity of one or many types is not given until the class or the method is declared and instanciated. Their usage allow one to gain performance and to secure types during compilation (generics avoid using casts which can be dangerous).

The most common use of generic classes is with collections such as sorted lists, hashtables, stacks, queues, trees and other collections where operations such as addition and deletion of elements of the collection are executed in a similar manner independently of the type of data that are stored. Here is an example of data that you often use and that uses generics, the List type :

```
// Declaration of a list containing integers
List<Int> my_list = new List<int>();
// Declaration of list containing ACDC
List<ACDC> DreamTeam = new List<ACDC>();
```

We can see here that lists of any type can be created using generics. More concretely, generics can be applied to:

- a class
- a structure
- a method
- an interface
- a delegate

It allows you to say that your generic thing can be of any type, by value or by reference. It is then possible de factorize lines of code and to avoid useless efforts.

2.2 Create generic classes

Today, we are going to the other side of the template to learn to make generic classes ! To describe a generic class, simply explain that it depends on non fixed type.
Here is an exemple that explains how to do that:

```
// Depends on an unknown type that we name T
public class MyBox<T>
{

      // Use of T as a normal type
      public T value;
      // Takes an parameter of type T
      public MyBox(T val) { value = val; }
      // Returns a value of 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()); // displays "2"
      Console.WriteLine(second_box.getValue()); // displays "two"
}
```

Very easy, isn't it ?

2.3 Constraints

When you define a generic class, you can apply constraints to the kind of types that the user can use for the parameters types when he instancies your class. If the user tries to instanciate his class with a type that is not authorized by a constraint, there will be a compilation error. These restrictions are called constraints. Constraints are specified thanks to the keyword **where**.

There are many kinds of constraints. Here are some of them:

- class: the type argument must be a reference type, including any class, interface, delegate or array type.
- <interface name>: the type argument must be, or implement, the specified interface. Many interface constraints can be specified. The interface that requires the constraints can also be generic.
- <class name>: the type argument must be, or inherit[1], from the specified class.

There are other constraints[2]. We have listed here the most important ones.

Here is an example of constraint:

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

Here, the Y type must respect the IComparable interface[3].

There can also be many constraints for one type: they are separated by commas.

3 Interfaces

3.1 Principle of interfaces

Interfaces are a very important concept of object oriented programming. It consists in defining a group of functionalities like methods, properties, events and indexors. Classes or structures that implement an interface must imperatively implement every method and property defined in the interface.
We use the "interface" keyword to declare them. There is no function body and we cannot instanciate an interface even if it is close to a class. A class and a structure can implement many interfaces at once. An interface cannot contain constants, fields, operators, constructors, destructors or types. Nor can it contain static members. Members of an interface are automatically stated as public and they cannot contain any access modifiers.

3.2 IEnumerable

C# offers lots of interfaces, most of which use generics. Do not hesitate to get informed about it. Here is an example of a sexy C# interface: IEnumerable is an interface allowing the user to browse a container. It allows amongst other things the use of an object in a "foreach" loop. The only method to implement in order to comply with this interface is:

```
IEnumerator GetEnumerator()
```

3.3 How to implement its own interface?

Nothing prevents you from defining your own interface. By convention, its name begins with an "I" and the first two letters are capital letters. Example:

[1] Heritage &co
[2] See "Go deeper"
[3] See the "Interfaces" part below

```
interface IDimensions
{
      float getLength();
      float getWidth();
}
class Box : IDimensions
{
      float lengthInches;
      float widthInches;

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

       // Implicit but mandatory implementation of IDimensions
      float getLength() { return lengthInches; }
      float getWidth() { return widthInches; }

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

           // Displays the box dimensions by calling methods of the object
           Console.WriteLine("Length : {0} ", box1.getLength());
           // displays "Length : 30 "
           Console.WriteLine("Width : {0} ", box1.getWidth());
           // displays "Width : 20 "
      }
}
```

Here is the same example but with an explicit implementation of the interfaces:

```
interface IDimensions
{
      float getLength();
      float getWidth();
}
class Box : IDimensions
{
      float lengthInches;
      float widthInches;

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

       // Explicit but mandatory implementation of 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 Exercise

Create an "Interface" project in an "Interface" directory.

- Create an ISport interface with at least four functionalities of your choice.
- Create the Volleyball, Football and Rugby classes in such a way that they implement ISport.
- In your Main(), do some tests with instances of the three classes.

4 Delegates

Delegates is a type that defines a method signature (and its return type). When you instanciate a delegate, you associate the instance of your delegate (e.g. the variable) with a method whose signature is the same as the one defined in the delegate.

Here is what it looks like:

```
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));
    }
}
```

Copy this code, look at what it does and try to understand why.

You must note one thing: there are two functions, square and minusFortyTwo. These two functions have the same signature and the same return type. They do not do the same thing but the same number and type of arguments and the same return type.

On the first line, we create a new delegate named MyDelegate. Then, in the main() function, we instanciate this delegate by creating the "intFunctions" variable. We assign to this variable the *square* method (without parenthesis - we are not trying to call the function). Then we call our delegate exactly like we call a function.

```
intFunctions(x)
```

This calls the method[4] that is in the intFunctions delegate. Once your delegate is instanciated, it can be used like a function.

In order to understand better what a delegate is, make your own tests.

What is the purpose of a delegate?

Here is an example. You create a class to represent your favorite data structure (list, tree, Fibonacci's heap) with a sort function in this structure. But you are a great programmer and so you made different algorithms to sort your data. When you call the sort function, you might want to choose the algorithm that has to be called. You could do the following :

[4] An instance of delegate can contain several methods. Indeed, you can write *intFunctions = minusFortyTwo;* then *intFunctions += square;* See what it does.

```
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
}
```

But as soon as you add a sort algorithm, you have to change the enum and add an if condition. With delegates you can do the following:

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

Here is how to replace all these ugly ifs which are a pain to write. Of course, you can find a lot of usages for the delegates. Sometimes they are not necessary but their use will simplify your life.

5 Functions

5.1 Variadic functions

Since you have started to learn C# , you now know how to create functions with arguments. But do you know how to create functions with a variable number of arguments ?
Let's take an example : you want a function which is going to write in a file (with a Stream object) several objects. In order do this, you could write a function like this one :

```
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);
}
```

As you can see, this would not be very handy if you wanted to write several times in a file. Obviously you could change the "object l" by an "Object array(object[] l)", and then iterate on it, but this would force you to create an array before you could call the function.

This is where the adjective "variadic" makes sense. The goal of this kind of function is simple : allowing you to call a function with a variable number of objects. For those who have already written some pieces of code in C, it should remind you of the "printf" function.

To achieve that, we need a new keyword : **params**. Here is the definition and an example from 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);
}
```

Here is the way to use the params keyword : "(params typename[] name)". You can call the function either with a "typename" array or with arguments of type "typename", separated by a comma.

In UseParams2, you should see a new keyword, **object**, which could actually represent every possible type.
It is, in fact, what we call an "alias" (a shortcut, if you prefer) for the Object class. This class is the mother of everything, the son of your son, the master of the gods, and even your old grandma's mother. Seriously, now that you know how inheritance works, we can tell you the truth: every class inherits, directly or not, from the Object class. You will find more information on MSDN or with your ACDCs (again). That's why you can write the object keyword to symbolize every type. In the first example, where we wrote "Stream stream = new FileStream(...)", the compiler didn't detect errors, because FileStream inherits from Stream, so the cast from a FileStream to a Stream doesn't bother us. (Please note that it's not true in the other way. It's up to you to find out why).

5.2 Anonymous functions

Well, now that everything is clear (or not) for you, try to call UseParams2 with the parameter "new int[3]1, 2, 3". Try to understand why the output is not what you expected.

Let's see what anonymous functions are. As you can guess, an anonymous function is a function... with no name! It does not help much, and what is the purpose of an unnamed function : how can we call it if it has no name?

Let's see what MSDN says about it.

> An anonymous function is an "inline" instruction or expression that can be used everywhere where a delegate type is expected. You can use it to initiliaze a named delegate or to give it as a method parameter instead of a named delegate.

MSDN

Just remember that an anonymous function can be used everywhere where a delegate is expected.

Let's go deeper to see two different types of anonymous functions: lambda expressions and anonymous methods. Note the difference between the *anonymous functions* and *anonymous methods*.

5.2.1 Anonymous methods

Anonymous methods allow us amongst other things to give a code bloc as a delegate parameter, which means that you can create the method that is encapsulated by your delegate directly in the instanciation of this one.

Here is an example:

```
// Creation of a delegate
delegate void Del(int x);

// Instanciation of the delegate with an anonymous method
Del d = delegate(int k) { /* your code here*/ };
```

You can write in your anonymous method (where there is *your code here*) as if you would code a function taking an int as a parameter and returning nothing. As you can see, the syntax is quite simple.

You may not see the point of this for now, but once you get used to delegates, you will be happy to use anonymous methods (and functions generally) instead of creating explicit ones. It is faster and sometimes more explicit.

Be careful - do not overuse the anonymous method and do not replace all of your methods by anonymous ones (and we don't think you want to do this). Use them wisely.

5.2.2 Lambda expressions

Lambda expressions: let's see an explanation from MSDN.

> 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.

If you do not understand the part explaining the tree types or LINQ, this is normal, and do not worry - we won't use these notions during this practical work.

The syntax is again quite simple[5] :

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

If you have only one parameter, then the parentheses are optional. The parameters' type is also optional in some cases (when the compiler cannot find the parameters' type alone).

```
() => 2 // no parameter
(x, y) => x == y // no need of the types of x and y
(int x, string s) => s.Length > x  // you need to specify the types of x and s here
```

[5] It should remind you of the *function* keyword in Caml

6 Exercises

6.1 Generic dynamic array and delegates

In this part of the practical work, we will implement our own data structure : a dynamic array. Dynamic means that its size can change. If we reach the maximum size of the array, we reallocate a bigger size for the array.

As you learned new notions during the reading of this TP, we will put them into practice: we will implement a generic dynamic array, using interfaces, delegates and anonymous functions.

We will give you the methods to implement, their signature, what they have to do, what notion to use... and you do the rest!

Our class will be called MyGenericTab. Be careful! It must be generic, and we require a constraint on the type: it has to inherit from the IComparable interface.

It will have the following attributes:

- An array of type T non initialized
- An integer containing the current number of elements in the array

And methods:

- A constructor: initiliazes the array and also its current number of elements

```
public MyGenericTab<T>() { ... }
```

- Insert: adds an element to the first free position of the array and if the array is full, we reallocate by doubling the size of the array.

```
public void Insert(T elt) { ... }
```

- Delete: removes the element given as parameter of the array, and shifts all the elements so that there are no holes. Returns true if the removal succeeded, false if not.

```
public bool Delete(T elt) { ... }
```

- At: returns the value contained at position i. Throws an exception if the index is invalid.

```
public T At(int i) { ... }
```

- mapDelegate: delegate to use for the map function. It has to be declared outside of the class.

```
public delegate T mapDelegate<T>(T elt);
```

- Map: applies the delegate given as parameter to all of the array's elements.

```
public void Map(mapDelegate funct) { ... }
```

- Sort: sorts the array by using the CompareTo method from the IComparable interface.

```
public void Quicksort(int left, int right) { ... }
```

- Print: prints in green the elements for which the predicate given as parameter are true, and in red if not.

```
public void Print(/* delegates that returns a boolean and takes as parameter
                                a T element */) { ... }
```

6.2 Bonii

- Improved Insert: the insertion is done by respecting the order of the array.
- Improved Delete: if only a third of the array is used, we reallocate the array by dividing its size by 2.
- Select: find what it is!

7 Deeper and deeper...

This part is here for those who want to know more because what we have shown here does not cover all the functionalities of the notions in C#.

We only show you a few features that exist. If you want to know more, Google is your friend!

- Value type
- Reference type
- Default keyword for genericity
- Safe types during compilation thanks to genericity
- new() constraint on a generic parameter
- We cannot implement an interface member explicitly: void ISampleInterface.SampleMethod(), but it is not so easy...