Practical C#12 : Genericity

1 Guidelines

At the end of this tutorial, you have to submit an archive that respects the following architecture :

```
rendu-tp-login_x.zip
|-- login_x/
    |-- AUTHORS*
    |-- README
    |-- Matrix
        |-- Matrix.sln*
        |-- Matrix*
            |-- Everything but bin/ et obj/
    |-- Tree
        |-- Tree.sln*
        |-- Tree*
            |-- Everything but bin/ et obj/
```

Of course, you must replace "`login_x`" by your own login. All the files marked with an asterisk are *MANDATORY*.

Don't forget to verify the following before submitting your work :

— The `AUTHORS` file must be in the usual format (a *, a space, your login and a line feed).
— No `bin` or `obj` directory in your project.
— **The code must compile !**

2 Introduction

The goal of this pratical is to introduce you the concept of genericity, it allows you to write more general and cleaner code, you'll be able to simplify your program's structure and avoid code duplication.
You'll see the following concepts : *generics*, *overloading*, *delegates*, *anonymous functions*, *lambda expressions*. These are really important concepts and are mandatory for any good programmer, please ask questions to your ACDCs if there's something you don't understand.

3 Course

3.1 Overloading

Function overloading is really simple and you probably have already used it. It allows you to have many functions or methods having the same name but taking different arguments. It can also be done on operators but we'll see that later.

Here are some examples :

3.1.1 Functions overloading

Here we want to code a *length* function so that it can calculate the number of characters of a *string* or the number of digits of an *int* without having to call a specific function for each different type (that would be the case in C but you'll see that next year)

```
static void Main(string[] args)
{
        Console.WriteLine(length(48984)); // output : 5
        Console.WriteLine(length("Hello World!")); // output : 12
}

static int length(int elt)
{
        int total = 0;
        while (elt != 0)
        {
                elt /= 10;
                ++total;
        }
        return total;
}

static int length(string elt)
{
        int total = 0;
        foreach (char c in elt)
                ++total;
        return total;
}
```

Note 1 : Constructors being just special methods, it is also possible to overload them.

Note 2 : Overloading is not determined by the return type, it doesn't help the compiler to distinguish 2 different functions.

3.1.2 Operator overloading

When you think about it a little bit, operators like + or - are functions, we have just added some syntactic sugar to allow an in-order notation like in mathematics. Indeed, in Lisp for example, the in-order notation is not native, so to compute an addition you need to write $+\ x\ y$ to do $x + y$, so we can clearly see that it's the + function applied the x and y parameters. Now that we know that operators are just simple functions, why not try to overload them ?

Here's an example where we want to create a matrix object, and we want to be able to substract 2 matrices. We could add a function *matrix_substract* that would take 2 matrices as parameters and return the total matrix, but it's not really intuitive, we want to be able to write *total_matrix = matrix_1 - matrix_2*.
But C# cannot do it automatically, we'll have to tell him how to proceed, and for that we just need to overload the minus operator with the *operator* keyword.

```
class Matrix
{
        public int[,] table = new int[4,4];

        public Matrix(){} // creates an empty matrix

        public Matrix(int nb)
        {
                // fill table with nb
        }

        public static Matrix operator -(Matrix m1, Matrix m2)
        {
                Matrix result = new Matrix();
                for (int i = 0; i < m1.table.GetLength(0); ++i)
                        for (int j = 0; j < m1.table.GetLength(1); ++j)
                                result.table[i,j] = m1.table[i,j]
                                                    - m2.table[i,j];
                return result;
        }

        public void print()
        {
                // ...
        }
}
```

```
class Program
{
        static void Main()
        {
                Matrix matrix1 = new Matrix(3);
                matrix1.print();

                Matrix matrix2 = new Matrix(2);
                matrix2.print();

                (matrix1 - matrix2).print();
                // works & prints a matrix filled with 1
        }
}
```

Note 1 : You'll notice that the constructor is overloaded and thus has 2 variants, so that we can instantiate a matrix with 2 different methods : empty or initialized with a value.
Note 2 : The table attribute is *public* to simplify the example, do not hesitate to add getters and setters when applying this example yourself.

3.2 Generics

For those who have already done some C++, the template notion should remind you something, it also exists in C# under the generics name (same for java)

Note : For those interested, C# generics are less flexible than C++ templates, the differences between these two can be found here :
`https://msdn.microsoft.com/en-us/library/c6cyy67b.aspx`

3.2.1 Explanation

The main goal of generics is to treat different types or data structures the same way.
There again you have already used this notion wihtout knowing it. Indeed, lists use generics. You can create an *int* list or a *string* list and use it the exact same way just by indicating the type in <> when instantiating it.

Example : *List<int> l1 = new List<int>()* or *List<string> l2 = new List<string>()*

3.2.2 Example

Let's take our matrix class again, let's say we want to be able to create a *float* matrix.
Two solutions :

- Create a second class that we'll call *Matrix_float*, copy paste our previous code and replace all *int* occurences by *float* in a dirty way
- Modify our class using generics

```
class Matrix<T>
{
        public T[,] table = new T[4,4];

        public Matrix() { } // creates an empty matrix

        public Matrix(T nb) // creates a matrix filled with nb
        {
                // fill table with nb
        }

        public static Matrix<T> operator -(Matrix<T> m1, Matrix<T> m2)
        {
                Matrix<T> result = new Matrix<T>();
                for (int i = 0; i < m1.table.GetLength(0); ++i)
                        for (int j = 0; j < m1.table.GetLength(1); ++j)
                                result.table[i,j] = (dynamic)m1.table[i,j]
                                                  - (dynamic)m2.table[i,j];
                return result;
        }

        public void print()
        {
                // ...
        }
}
```

```
class Program
{
        static void Main()
        {
                Matrix<float> matrix1 = new Matrix<float>(3.5f);
                matrix1.print();

                Matrix<float> matrix2 = new Matrix<float>(2.7f);
                matrix2.print();

                (matrix1 - matrix2).print();
        }
}
```

As you can see, we didn't have to change much, we just added *<T>* to *Matrix* each time it designated a type (so everywhere apart from the constructors), and we replaced all *int* keywords by *T*, nothing more.

3.2.3 Nothing more ?

Well not really, some may have seen that the keyword *dynamic* appeared when substracting the matrix values. Indeed, without that keyword visual studio refuses to compile saying that the minus operator cannot be applied to operands of type *T* and *T*.

So if you understood well the previous part on operator overloading, Visual is telling us : I can't find any operator overloading on *minus* for type T (for those who are lost : I don't know how to subtract 2 objects of type T).
And, indeed, it's completly logical, T can represent anything, it be be a car object and in that case it would have no meaning to substract two cars !
So the solution would be to overload the minus operator for the T type ? Ok great idea but how ? As T is general we have absolutly no idea on how to compute a substraction.

The real answer is that we cannot know of what type T will be at compile time, and thus we're going to explicitly say to the compiler with the *dynamic* keyword that we want to check the ability to subtract two objects of type T at runtime and not at compile time. Indeed, at runtime T is replaced by its real type and so we know if we are subtracting *int* or *cars*.
Let's test that with an example :

```
class Program
{
        static void Main()
        {
                Matrix<string> matrix3 = new Matrix<string>("Hello");
                matrix3.print();

                Matrix<string> matrix4 = new Matrix<string>("World");
                matrix4.print();

                // Until then, matrices are displaying,
                // generics are perfectly working

                (matrix3 - matrix4).print(); // the program crashes
        }
}
```

If you followed everything, you should understand that we cannot subtract objects of type *string*,thus it crashes at runtime and an exception indicating that the minus operator cannot be applied to *string* is raised.
This last part being complex, we won't expect from you to get everything at the first reading, don't just read but try to code the example on your own computer and do not hesitate to ask you ACDCs for help.

3.2.4 Enter the Matrix

Explain to your ACDCs if the following code is working and why in a precise and simple manner to potentially grab some bonus points :

```
class Program
{
        static void Main()
        {
                Matrix<Matrix<Matrix<int>>> matrix5
                  = new Matrix<Matrix<Matrix<int>>>(
                    new Matrix<Matrix<int>(new Matrix<int>(5)));
                Matrix<Matrix<Matrix<int>>> matrix6
                  = new Matrix<Matrix<Matrix<int>>>(
                    new Matrix<Matrix<int>(new Matrix<int>(3)));

                Matrix<Matrix<Matrix<int>>> matrix7 = matrix5 - matrix6;
        }
}
```

3.3 Delegates

Once again for those who have been practicing in C or C++, delegates should remind you of function pointers. For the others, let's see what it is :

3.3.1 Explanation

Delegates are types used to define methods. They permit the instantiation of variables representing functions. Hence, those *delegates* can be used to assign a function to another one. Which could be thought of as a reference to a function. Here is an exemple to clarify this point :

3.3.2 Example

You have seen earlier this year that there were different ways to sort a list of elements (if you don't remember, here is the wikipedia page : `http://en.wikipedia.org/wiki/Sorting_algorithm`)

You probably know about the *bubble sort* : easy to implement but not really efficient when it comes to sorting a big number of elements. Hence, what you can do here is to implement the list structure and choose the best fitting algorithm depending on the number of elements. If there are less than 5 elements we might want to use the bubble sort which is potentially more efficient on a small number of elements.

```
class MyList
{
        private List<int> list;

        public delegate void sort_method();
        public sort_method sort;

        public MyList(int size)
        {
                list = new List<int>(size);

                if (list.Capacity <= 5)
                        sort = bubble_sort;
                else
                        sort = quick_sort;
        }

        private void bubble_sort()
        {
                // ...
                Console.WriteLine("Bubble sort done");
        }

        private void quick_sort()
        {
                // ...
                Console.WriteLine("Quick sort done");
        }
}
```

```
class Program
{
        static void Main(string[] args)
        {
                MyList small_list = new MyList(4);
                small_list.sort(); // prints "Bubble sort done"

                MyList big_list = new MyList(70);
                big_list.sort(); // prints "Quick sort done"
        }
}
```

Nothing really complicated here : we create the delegate type *sort_method()*, and we initialize *sort* as public so that we're able to use it and finally we assign the sorting method that fits depending on the list's size in the constructor.

As you may have noticed *bubble_sort* and *quick_sort* are *private* and are not visible from the *Program* class, only *sort* is visible. Thus someone who isn't supposed to know the content of our *MyList* class doesn't have to think to know which one to use, you are the one who has the control.

Note : here *sort_method* has nothing in its parentheses, it's obviously because *bubble_sort* and *quick_sort* have no parameters.

3.3.3 Anonymous functions

Anonymous functions allow to create thanks to delegates functions nested in other functions. It allows to have a better code encapsulation to make it cleaner. Indeed, a function 1 declared in a function 2 doesn't exist outside of function 2 and doesn't pollute its namespace.

To go on on our example, we're going to add a sorting method which will be an intermediary solution between *bubble_sort* and *quick_sort* and which will be anonymous : it won't be visible or usable by any other method and will only exist through our *sort* variable.
So here's the new version of our constructor :

```
public MyList(int size)
{
        list = new List<int>(size);

        if (list.Capacity <= 5)
                sort = bubble_sort;
        else if (list.Capacity <= 20)
                sort = delegate()
                {
                        // ...
                        Console.WriteLine("My super secret anonymous
                                        function done");
                };
        else
                sort = quick_sort;
}
```

The rest of the class doesn't change.

3.3.4 Lambda expressions

Finally, the last concept for this practical : *lambda expressions*.
It is an extension of anonymous functions. Indeed, they allow you to simplify the code by removing the parameters' types and the *delegate* keyword. The previous anonymous function translated to lambda expression would be :

```
sort = () =>
{
        // ...
        Console.WriteLine("My super secret anonymous function done");
};
```

We were able to get rid of *delegate* using => after the parentheses.

One last example to make the difference between the two when there are parameters. Here is the function *triple* coded as an anonymous function

```
class Program
{
        delegate int my_int_del(int nb);

        static void Main(string[] args)
        {
                my_int_del triple_anonymous = delegate(int x) { return 3 * x; };
                my_int_del triple_lambda = x => 3 * x;

                Console.WriteLine(triple_anonymous(5)); // prints 15
                Console.WriteLine(triple_lambda(5)); // prints 15
        }
}
```

In this example you can see that the type of x is not declared when using the *lambda* notation. It allows you to use more genericity, write less code and hence to be clearer so do not hesitate to use it ! *Note* : be careful, don't believe that not writing the type of the parameter will allow you to use the *triple* function with any type, indeed *triple_lambda* is an *my_int_del* type and thus has an *int* return value, so it's impossible to use *triple_lambda* with a *float* unless using generics.

4 Exercises

4.1 Exercice 1 : Matrix again

Code the *Matrix* class on the same model as the one in the course, you'll obviously have to use *generics* and will have to add the ability to multiply matrices with operator overloading.

4.2 Exercice 2 : Binary tree

Data structures are one of the most obvious applications of genericity. Hence, in the exercise part you will have to implement one of the data structures you have seen in class : the binary tree. Obviously, you will have to use what you have learned just before.

The first step will be to create the *Tree class*, using *generics*.

```
class Tree<T>
{
        // ...
}
```

4.2.1 Attributes

As you may know, each node of your tree will be composed of an element of type T, a right and a left child. In addition to this, your structure must have a *delegate* of the type :

```
delegate int CompType(T a, T b);
```

This delegate will represent the comparison function between two elements of type T. If the function returns a positive number then *a* is "bigger" than *b*, if it returns *0* then *a* is equivalent to *b*. This functions will allow you to organise a binary search tree.

Hint : In order to simplify traversal functions you could implement a subclass Node containing le node value and its sons. Therefore, you would only keep the root node as an attribute in your Tree class.

4.2.2 A Constructor

Add to your class a simple constructor, that will instantiate a tree from a node and its comparison function.

```
public Tree(CompType<T> comp, T value);
```

4.2.3 Traversal Functions

You will have to code 3 traversal functions. The first two ones are the classical ones the depth first and breadth first traversal :

```
delegate void MapFunction(ref T elt);
public void depth_first_traversal(MapFunction function);
public void breadth_first_traversal(MapFunction function);
```

The third function will be a bit different. This function will traverse the tree searching for an element : it should recurse to the right if the element is bigger than the current node and to the left if it is smaller. Then, two different situations can occur : the element is in the tree and you should apply the function *equal_case* and if it is not you should apply the *null_case* one. Remember that you must use the tree's comparison function to recurse in the correct direction. As you can implement the class the way you want, you are free to modify the prototype of this function, that will only be used inside of the Tree class.

```
private delegate bool BinFunc(ref Node node, T val);
private bool binary_traversal(ref Node node,
                              T val,
                              BinFunc null_case,
                              BinFunc equal_case);
```

4.2.4 Tree operations

When using a Tree structure, three functions are really important : insertion, deletion and element search. Hence, you will have to implement each one of them, using the previously coded *binary_traversal*.

```
public bool insert(T val);
public bool find(T val);
public bool delete(T val);
```

For *find* and *delete* you will have to return *true* if the element was found, *false* otherwise. For the insertion, you should not add the element to the tree if it is already present and you should return *false*. Although insertion and searching functions must be pretty straight forward, the deletion one might be harder to implement. You will have to handle different cases after finding the element. If the node doesn't have a right child you should just assign it its left child value and if it does not have a left child, the value of its right child. Then if none of its children are null, you should assign it the value of the bigger node in its left subtree. To do so you should find the right most element of this subtree and delete it as well. If this last part doesn't seem clear, you are free to find inspiration in your algorithm course.

4.2.5 A new constructor

Now that you have coded the *insert* function you will be able to easily add elements to your tree. Add a new constructor that will take an array of elements as parameter and will add them all to the tree.

```
public Tree(CompType<T> comp, T[] values);
```

4.2.6 Complexity

In order to demonstrate the interest of binary search trees, you will have to compare the difference of performances with lists. To do so you can use a variable that you increment in the anonymous/lambda function you will pass to the list *find* function and to the constructor of your tree. You can try with different configurations (ordered lists with binary search, etc.) but always try with a lot of values to avoid unprecise results.

4.2.7 Bonii

As you might have realized the exercise part of this practical is quite short. However it is important that you understand the concepts you have been taught. Firstly, about the genericity that wil allow you to save a lot of time while coding, but also about the Tree structure, which is commonly used in computer science. Here are some ideas of bonuses you might want to implement to improve your grade.

— AVL : Add the different rotation functionalities during insertion to optimize search (if the elements are added in order in our Tree, our implementation of it will create a *comb* tree which is really inefficient for search).

— Implement an arithmetic operation solver using a tree. Handle the parenthesis.

The Bullshit is strong with this one...