TP C#12

1 Submission rules

You must submit a zip file with the following architecture :

```
rendu-tp-firstname.lastname.zip
    |-- rendu-tp-firstname.lastname/
        |-- AUTHORS
        |-- README
        |-- Matrix/
            |-- Matrix.sln
            |-- Matrix/
                |-- Tout sauf bin/ et obj/
        |-- Tron/
            |-- Tron.sln
            |-- Tron/
                |-- Tout sauf bin/ et obj/
```

You replace *firstname.lastname* with your login.

Don't forget to check the following rules before submitting your practical :

— The file AUTHORS must be in the usual format : * *firstname.lastname$* where the character '$' is a line break.

— You must follow the subject **scrupulously**, especially **respecting the prototype** of each function.

— No folder bin and/or obj in the project.

— **The code must compile !**

2 Introduction

2.1 Objectives

In this practical, you will see the following notions :

— operator overload

— Generics

3 Course

3.1 Overloading

You've already seen and used overloading in the practicals but this practical will do a little review. Method overloading is really simple. It allows you to have many methods with the same name but taking different arguments. Let's see an example with the method `MiniCat` of TPCS6 :

```
static void MiniCat(string InputFile);
static void MiniCat(string InputFile, string OutputFile);
```

Or more simply you already use overloading everyday with the `WriteLine` method in the `Console` class :

```
public static void WriteLine(bool value);
public static void WriteLine(char value);
public static void WriteLine(double value);
public static void WriteLine(int value);
```

3.1.1 Operator overload

When you think about it a little bit, operators like + or - are methods. 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 is the + function applied the x and y parameters. Now that we know that operators are just simple methods, why not try to overload them ?

Here is an example where we want to create an object `Vector`, and we want to be able to subtract 2 vectors. We could add a method `VectorSubtract` that would take 2 vectors as parameters and return the total vector but this is not really intuitive. We want to be able to write `vectorFinal = vector1 - vector2`. C#, however, cannot do it automatically. We'll have to tell it how to proceed, and for that we simply need to overload the minus operator with the `operator` keyword.

Let's see an example with our class `Vector` :

```
public class Vector
{
    int x;
    int y;
    int z;

    public Vector()
    {
        this.x = 0;
        this.y = 0;
        this.z = 0;
    }

    public Vector(int x, int y, int z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public static Vector operator -(Vector v1, Vector v2)
    {
        Vector result = new Vector();
        result.x = v1.x - v2.x;
        result.y = v1.y - v2.y;
        result.z = v1.z - v2.z;
        return result;
    }
}
```

You will notice that the constructor is overloaded and thus has 2 variants, so that we can instantiate a vector with 2 different methods : empty (coordinates initialized to 0) or initialized with a value.

You will also notice that with the overloading of the minus operators we just have to launch `vector1 - vector2` to do the subtraction.

3.2 Generics

For those who have already done some C++, you may recall the notion of the `template`. It also exists in C# under the `generics` name 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 Explication

The main goal of generics is to treat different types or data structures in the same way. You have already used this notion without 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>()
List<string> l2 = new List<string>()
```

3.2.2 Example

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

— Create a second class that we will call `Vector_float`, copy paste[1] our previous code and replace all `int` occurrences by `float` in a dirty way

— Modify our class using `generics`

Obviously we will take the second option. To do that we just have to say that we will use a vector of the general type `T` (`T` is a common name-you can call it whatever you want). Now, you just have to replace all the occurrences of `int` by `T`.

1. copy paste, it is bad !

Example :

```
public class Vector<T>
{
    T x;
    T y;
    T z;

    public Vector()
    {}

    public Vector(T x, T y, T z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public static Vector<T> operator -(Vector<T> v1, Vector<T> v2)
    {
        Vector<T> result = new Vector<T>();
        result.x = (dynamic)v1.x - (dynamic)v2.x;
        result.y = (dynamic)v1.y - (dynamic)v2.y;
        result.z = (dynamic)v1.z - (dynamic)v2.z;
        return result;
    }

    public override String ToString()
    {
        return x + "," + y + "," + z;
    }
}
```

As you can see, we did not have to change much, we just added `<T>` to `Vector` each time it designated a type (so everywhere except for the constructors), and we replaced all `int` keywords by `T`, nothing more.

3.2.3 Nothing more ?

Not really, some of you may have seen that the keyword `dynamic` appeared when subtracting the vector values. Indeed, without that keyword monodevelop 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, monodevelop is telling us : I cannot 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 is completely logical. `T` can represent anything. For example it could be a car object and in that case it would have no meaning to subtract two cars ! So would the solution be to overload the minus operator for the `T` type ? OK great idea, but how ? As `T` is general, we have absolutely no idea about how to compute a subtraction.

The real answer is that we cannot know of what type `T` will be at compile time, and thus we are going to say to the compiler explicitly 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.
CAUTION : this does not resolve the problem. Indeed, if the type `T` does not have an `-` operator overloaded it will crash when you execute it. It can be resolved in C# with some constraints added to the type. It can be done with the keyword `where`. We won't tell you about it in this practical but you can have some information, if you want, on this link :
`https://msdn.microsoft.com/en-us/library/bb384067.aspx`

3.2.4 It is not finished !

Very last information in this practical : in the class `Vector`, you can see that a method `ToString()` exists. You have to know that every class in C# inherits implicitly from the class `Object`. This class contains some methods that we can override (so we have to use the `override` keyword). The example, here, is with the method `ToString()` that returns a string representing the actual class (so we can display it). For more information you can check this website :
`https://msdn.microsoft.com/en-us/library/system.object(v=vs.110).aspx`

4 Exercises

4.1 Enter the Matrix

This part needs to be coded in the project `Matrix`.
Let's begin with a little warm-up with the implementation of matrix in C#. We speak here about the mathematics definition of matrix.

To do it, you have to create a generic class `Matrix` that will have only one attribute : a double dimension array.

4.1.1 Constructor

First, if we want our class to work, we need a constructor. For us, only one constructor is too easy so you have to code three constructors.

```
public Matrix(int dim);
public Matrix(int height, int width);
public Matrix(int height, int width, T init);
```

The first will use `dim` to create a square matrix and will not assign values in the array.
The second will use the two parameters to give dimensions of the array and it will not assign values in the array.
The third will do the same thing as the second, but it will assign the whole array with the `init` value.
You have to throw an `ArgumentException` with an appropriate message when the dimensions are negative.

4.1.2 Addition/Subtraction

Now, let's continue the warm-up with the operations on our matrix. You have to implement the addition and subtraction for a matrix. Of course you have to overload -/+ operators.
You have to implement the following methods :

```
public static Matrix<T> operator -(Matrix<T> a, Matrix<T> b);
public static Matrix<T> operator +(Matrix<T> a, Matrix<T> b);
```

Of course you have to check the dimensions of the two matrices and throw an `ArgumentException` if it is not possible to do the operation (different length or width).

4.1.3 Multiplication

Now you have to implement the multiplication :

```
public static Matrix<T> operator *(Matrix<T> a, Matrix<T> b);
```

We are kind that are giving you the formula :

$$\forall i, j : c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} b_{nj}$$

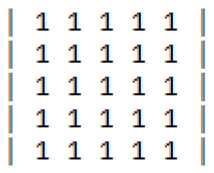
Like the other operations you have to check the dimensions and throw an `ArgumentException` if needed.

4.1.4 Display

You have, now, to display your matrix !
To do that you have to override the method `ToString` as you read in the course (which you have already read, of course !).

```
public override String ToString();
```

Each line needs to begin and end by '|'. Before and after each element of the matrix there will be a space.
Let's see an example with a 5x5 matrix where all the elements have the value 1 :



```
| 1 1 1 1 1 |
| 1 1 1 1 1 |
| 1 1 1 1 1 |
| 1 1 1 1 1 |
| 1 1 1 1 1 |
```

4.1.5 Accessors

We want to access to all the values of our matrix without access directly to the attributes. To do that we want to do directly `matrix[i, j]`. Thanks to C#, we can do it simply using the properties. To do that you just have to fill the following property :

```
public T this[int i, int j]
{
    get
    {
        //FIXME
    }
    set
    {
        //FIXME
    }
}
```

For a better understanding of this property, let's take an example with a class `Array` which has only one attribute : a one dimensional array named `arr`. We want to access to the array but without a method. We especially want to do `object[i]`. In this case the property will be :

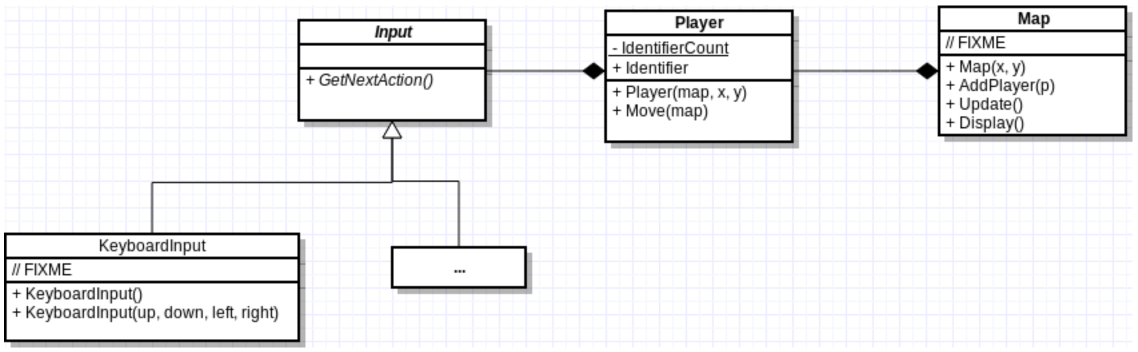
```
public T this[int i] //You can add  an undefined
{                     //number of variables
    get
    {
        return arr[i];
    }
    set
    {
        arr[i] = value;
    }
}
```

Now it is your turn !

4.2 Tron

Well, we could continue working on many little exercises about overload and generics but we will be nice. This week's project will be something entirely different! You will code a Light Cycle game. You have to do it in the Tron project.

Your project will follow this UML diagram :



4.2.1 Some important points

For this part, you will be less guided than usual. You must implement each requested method, but it will be easier to add auxiliary attributes and methods. You must use the most restrictive viability : an auxiliary method used only in its own class must be private.

Overall, the subject will often have little explicit advice about the method you should follow. As long as the expected result is clear, take some time to think about the possible approaches before asking for help.

4.2.2 The rules

The map starts empty, with the players deployed on it. For each game iteration, each player moves in one way, the same as the one before if no action took place.

At each move, players leave a "wall" behind them. If a player tries to move to a wall or go off of the map, he is removed from the game. The winner is the last one left.

4.2.3 Input

Let's start with something simple. We need to manage user inputs! We want our game to work with any type of input, keyboard or others. First, we need to represent the different inputs. Let's just use an enum for that.

To manage any kind of input in a generic way, we will use an abstract class with a single method : `GetNextAction`. It returns the player's next action. The purpose of this class is to manage any kind of input regardless of how it works or what it needs.

```
public abstract class Input
{
        public enum Action {up, left, right, down};
        public abstract Action GetNextAction();
}
```

4.2.4 KeyboardInput

For this project, you will need to use at least keyboard inputs. You have to create a class doing this, inherited from `Input`. It can manage any key to represent the directions up, down, left and right. If the player hasn't pressed any key since the last cycle, the last action must be repeated.

You have to implement two constructors : the first one takes four keys as parameters to bind keys to actions, and the second one affects the arrow keys by default.

Be very careful about the order of the parameters.

```
public KeyboardInput(ConsoleKey up, ConsoleKey down,
    ConsoleKey left, ConsoleKey right);
public KeyboardInput();
```

This class isn't easy to implement properly, and we don't want you to be stuck at the beginning of the project. We've defined some thresholds. They're all *mandatory* up to the third one included, the rest is bonus. You can stop on a threshold to progress on the project, and come back to finish it once the rest is done. You cannot use the content of `System.Windows.Input`.
The thresholds are as follow :

1. Keyboard inputs are read. Don't stop here.
2. The game doesn't pause at each cycle.
3. Several KeyboardInputs can be used in parallel for several players.
4. The case when different players press a key on the same cycle must be managed properly.
5. KeyboardInput no longer needs code to be written outside of the class, for example in the main game loop.

Hint : if you're lost, do the *opposite* of the bonus thresholds.

4.2.5 Player

This class represents a player. It must have a position, and an unique identifier. This identifier must be assigned directly on the instantiation. Think about static members.

When a player tries to move, we need to check if he has crashed, which happens when he tries to go off the map or on a wall left by a player. You also need to mark the map to indicate that there is now a wall here.

Each player has his own color. It applies to the display of the player itself, as well as its walls. You can manage these colors either in the Player class, or in the Map class. Or something else, up to you. You need distinct colors for the first 4 players at least ; additional players may share a common color.

You need to implement at least the following methods :

```
public Player(Input input, int x, int y);
public bool Move(Map map);
```

4.2.6 Map

This is the project's main class. It has to manage each player and the state of the map itself. `Update` must move each player, check if he is still alive, and look for the end of the game. The return value is the identifier of the winning player, 0 if the game isn't over.

For the display, you need to display the players, the walls, and the map borders. Separate the logic from the display, don't touch `Console.Write` in the method `Update`, and vice versa.

You need to implement at least the following methods, but we strongly recommend that you do more than that. We forbid functions that are longer than 50 lines. Factorize your code and separate it in little methods that make sense.

```
public Map(int x, int y);
public void AddPlayer(Player p);
public int Update();
public void Display();
```

4.2.7 Main

The Main will initialize the map, add players and run the game's main loop. By default, the game runs with two players, one using ZQSD and the other the arrows.

Running the game normally without argument must directly run the game. If you want to ask the user for map dimensions, controls and so forth, do it through the program parameters (you know, the arguments of the Main function). The best would be a config file.

4.2.8 Impress us !

A lot of notions have been covered today. That is why this practical is not too long. We expect to get many bonuses from you ! You can add classes, methods or some inputs (mouse, network ...). Update the quality of display or inputs. All the bonus as you have made must be on your README otherwise it won't be marked.

The Code is the Law.