TPC# 5

1 Submission guidelines

You must respect the following architecture :

```
rendu-tp-prenom.nom.zip
        |-- rendu-tpcs5-prenom.nom/
                |-- AUTHORS
                |-- README
                |-- Explorer/
                        |-- Explorer.sln
                        |-- Explorer/
                                |-- All but bin/ and obj/
```

You must replace *prenom.nom* by your login. Don't forget to check :

— The AUTHORS file has to be of the form : ** prenom.nom$* where $ is a linefeed. You should not write $ in your AUTHORS file.

— The README file should contain a description of the problems you encountered on the subject and the bonus you did.

— You should follow the subject **closely** and respect the functions **prototypes**.

— **No bin and obj folder in your submission.**

— **Your code must compile !**

2 Before we start

In this practical you will discover a lot of new features that are not always easy to master. You will find this subject harder than the previous ones : **Do not hesitate to ask the assistants for help and make sure you read the course part.**

3 Course

Today you will learn about object oriented programming in C# It is a programing *paradigm*, which means it is a representation of a model of the world : in other terms, it is another way of expressing ideas. (Functional programming, what you learnt with Ocaml, is another paradigm.)

3.1 Classes

Classes are a way of representing a data type using to properties. The point of classes is to regroup, using the manner, objects/concepts that have same properties. One can create instances of class that we will call **Objects**.

The classes' properties can be split into 2 groups :
— Attributes, which is the data
— Methods, which define a behavior.

Attributes. Are variables declared inside the Class. There are two types of attributes : Static, which are stored inside the class and have the same value for every class instance. The other type is the instance attributes, which are different for each class instance. Those are stored in the object.

Methods. Are functions/procedures defined inside the class. They are common to every instance of the calss and are linked to the object (the result will depend on the attributes of the instance) : They require a class instance to be called. They are static methods that are not linked to an object, you are already using them :

```
Console.WriteLine(string str);
```

Console is a class and WriteLine is a static method.

Example : Class declaration.

```
public class MyClass
{
        //you can declare attributes of any type
        //(even one defined by another class of yours)
        public static string myStaticAttribute = "A random value";
        public int myAttribute;

        public static string myStaticMethod()
        {
                //here the instructions
                //for example return myStaticAttribute
                return myStaticAttribute;
        }
        //of course the return type can vary
        public void myMethod()
        {
                //here the instructions
                //for exemple write myAttribute's value:
                Console.WriteLine(myAttribute);
        }

}
```

3.2 Objects

The creation of an object from a class is called **instantiation**. Classes make possible the instantiation of similar objects.

Constructors : A *constructor* is a special procedure that allows you to create an object. An empty constructor is provided by default but you can create your own and add parameters to it. You can execute instructions inside a constructor if you want to initialize your attributes for example.

```
public static int FlipNumber(int n);
```

Instantiation examples :

```
/*
remark : even if the constructor used here (by defaut)
does not take any parameters, we use parentheses
*/
MyClass myObject = new MyClass();
```

The default constructor does nothing, attributes won't be initialized and there will be an error when one tries to access it. We can create our own constructor that will initialize the value **inside the class**.

```
//Notice how the constructor should have the name of the class
public MyClass(int attr)
{
        //I can initialize my attributes the way I want here
        myAttribute = attr;
}
```

Once an object is created (and the attribute are initializedà, I can access them like this :

```
MyClass myObject = new MyClass(42);
myObject.myAttribute; //myAttribute is equal to 42
//same for the methods
myObject.myMethod(); //Prints 42
```

It is possible to access static methods and attributes without creating objects :

```
MyClass.myStaticAttribute;//"A random value"
MyClass.myStaticMethod();//here again the parenthesis because it's a method
```

3.3 Accessibility

It is possible to limit the access to some properties of the class thanks to *access modifiers* :

— **public** : properties that are accessible outside of your class (via an objects if it is an instance properties).

— **private** : properties that are accessible only from the class. They are not transmitted by inheritance.

— **protected** : properties are accessible from the outside only by inherited classes.

By default, all attributes and methods are private. If you want to access it from outside of the class (using an instance/object) you have to give it *public* or for the attributes create a method to access it or modify it. You can also create methods called accessors that will access or modify the attributes. Accessors must be public.

3.4 Inheritance

Inheritance is a key feature of object oriented programming. Inheritance allows the user to regroup attributes and methods that are common to multiple classes. To do so, one defines a class that contains those properties. This class is called base class. The classes that have these attributes will inherit it from the base class and define their own :

```
public class Vehicle
{
        public string brand;
        public int maxSpeed;
}
public class Car : Vehicle
{
        //define here your 'Car' class
}
public class Bike : Vehicle
{
        //define here your 'Bike' class
}
```

3.5 Structures

Structures are another way of defining a data type. Structures are very similar to classes, to declare a structure use the *struct* keyword :

```
public struct Point
{
        int x;
        int y;
}
```

You can declare structures the same way you declare variables :

```
Point p;
p.x = 1;
p.y = 2;
```

Unlike with classes, by default, structure members are public.

3.6 Enums

Enums are a practical way of defining multiple constants. To declare an enum, use the *enum* keyword :

```
enum Direction{ Left, Right, Up, Down};
```

You can then declare varibles of the type of your enum :

```
Direction way = Direction.Left;
```

3.7 Exceptions

Exceptions are *enum* when an instruction cannot be executed. You have already seen some :

```
int[10] myArray = { 0 };
int a = myArray[10]; //Out of bounds since arrays start at 0;
```

```
int result = 42/0; //impossible
```

To control the execution of your program and avoid errors due to exceptions, you have to use the **try catch** :

```
try
{
        int i = 0;
        result = 42/i;
}
catch (Exception e)
{
        Console.WriteLine("You cannot divide by 0");
}
```

The keyword **catch** is the code block that will be executed in case of an exception in the try part.

Exceptions are objects of the class Exception, you can define your own exceptions.

4 Practical : Explorer

You will now put into practice what you have learned in the course : The goal of this TP is to create a simple game in the console.

4.1 The rules

Le game has a two dimensional grid of finite height and width. You will control a character and the goal of this game is to finish a level : First, you will have to reach a point on the map. Then, you will implement enemies that you will have to kill.

The game will be played turn by turn, at each turn, you will take an action : First you will only ba able to move, then, you will be also be able to attack an enemy if it is next to you (left, right, up, down).

When you add enemies to the game, your character will lose health points each time it ends a turn next to an enemy. You lose if you don't have any health points left.

4.2 part 1 : the map/grid

Create a class *Map* to represent the map of your game, for the first step you will only need 4 attributes : a table of ints of 2 dimensions, the height, the width and a character :

```
public class Map
{
        char[,] map;
        int width;
        int height;
        Character hero;
        //int obj_x;
        //int obj_y;
}
```

Constructor Then add a constructor to your class. You must be able to create maps of different sizes, adjust your argument of the constructor.

```
public Map(/*put here arguments for the constructor*/)
{
        //initialize your class' attributes
}
```

Methods Your class *Map* must contain at least 3 methods :
— display() displays the map at the end of each turn.
— update() updates the map at each turn.

— is_over() to check if the player has ended the game by going to the end point, killing all the enemies or dying.

You can use the method *Console.Clear()* when you print a new map to avoid having the previous one on the screen.

Check your attributes and method accessibility

4.3 Part 2 : The character

You have to create a Character class for the player. The goal here is to interact with the character and to move around the map. Your Character will have at least the following attributes :

```
public class Character
{
        int life;//store the character HPs
        int x;//position in the x axis
        int y;//position in the y axis
}
```

Constructor You will create a constructor for your class. It would be a smart move to put health points in the argument so you will be able to change the difficulty of the game when you add enemies.

For now, you will only have to implement the method *move* that will move your character.

```
move(int dx, int dy);
```

Be careful, it is not here that you will ask the user for input. Here you will just update the attributes of the class.

Check your attributes and method accessibility

4.4 Part 3 : The main function

Once you have created your class Map and Character, it would be nice to make it work and play the game!

Your function *main* should be a pseudo-infinite loop that stops when the player ended the level. At each iteration, you will have to update the map and display it.

You will then have to detect the user action (key strikes) to move the character :

```
Console.ReadKey();
```

This function is blocking : it will stop the execution of the program until you press any key.

4.5 Part 4 : enemies

The last step is to add enemies that the player will have to avoid or eliminate to finish the game. To kill them, you will have to be next to it and hit enter (*ConsoleKey.Enter*). You will have to check all neighbor cells of your character when you move arround the grid.

Enemies To add enemies, create an enemy class. The class should inherit of Character and add methods and attributes that you need.

4.6 Conclusion

This practical is not easy. You are free to create the game as you want. Make sure to understand everything in the course part. This will be very helpful for the rest of the year.

You can add a lot of bonuses ! Show your assistants what you are capable of !

"The Code is the Law"