TP C#4 : Pokemon

Assignment

Archive

You must submit a zip file with the following architecture :

```
rendu-tpcs4-firstname.lastname.zip
|-- firstname.lastname/
    |-- AUTHORS
    |-- README
    |-- Exceptions/
        |-- Exceptions.sln
        |-- Exceptions/
            |-- Everything except bin/ and obj/
    |-- miniPokemon/
        |-- miniPokemon.sln
        |-- miniPokemon/
                |-- Everything except bin/ and obj/
```

— You shall obviously replace *firstname.lastname* with your login (the format of which usually is *firstname.lastname*).
— The code **MUST** compile.
— In this practical, you are allowed to implement methods other than those specified. They will be considered bonuses. However, you must keep the archive's size as small as possible.

AUTHORS

This file must contain the following : a star (*), a space, your login and a newline. Here is an example (where $ represents the newline and ␣ a blank space) :

```
*␣firstname.lastname$
```

Please note that the filename is AUTHORS with **NO** extension. To create your `AUTHORS` file simply, you can type the following command in a terminal :

```
echo "* firstname.lastname" > AUTHORS
```

README

In this file, you can write any and all comments you might have about the practical, your work, or more generally about your strengths and weaknesses. You must list and explain all the bonuses you have implemented. An empty `README` file will be considered as an invalid archive (malus).

1 Introduction

1.1 Objectives

The objective of this tutorial is to discover the Object Oriented Programming (OOP). Through this subject, you are going to learn notions like Classes and Inheritance. OOP is a paradigm : it is a way to solve an algorithmic problem. You already have seen the functional one with CamL but there is a whole lot more.
Furthermore, you are going to discover how to use Enumerations, Exceptions and Structures.

2 Course

2.1 Enumerations

Enumeration are a special type of variable. They represent a list of values. These values have different names in the code but are represented by integers. Therefore, it is possible to assign a specific integer to one or more of the values (see line 8 in following statement).

```
1  enum Color {
2      Blue,
3      Red,
4      Yellow,
5      Orange,
6      White,
7      Black,
8      Green = 10
9  };
```

We can see in the statements below, how to use the values of an *enum*. The easier and more explicit way is the first one. The two other statements are here to show you that values are represented by *integers*.

```
Color myColor = Color.White;
Color myColor = (Color)4; // myColor = Color.Orange
int n = (int)Color.Green; // n = 10
```

2.2 Structures

In C# , a *structure* is very similar to a class. A structure can contain several variables and even methods. It is better to use a *structure*, instead of a *class*, if need only need is to store variables. This will then represent a group of variables.
To define a structure in C# , simply do this :

```
public struct Point3D
{
    double x;
    double y;
    double z;
}
```

To initialize a *data structure*, you must do this :

```
Point3D p;
p.x = 10.0;   //X field assignment
p.y = 5.5;    //Y field assignment
p.z = -2.75;  //Z field assignment
```

2.3 Classes

To understand how classes work, nothing is better than an example.

```
public class Human
{
    //Public Attributes
    public string name;

    //Private Attributes
    private int age;

    //Getter & Setters
    public int Age
    {
        get { return age; }
        set { age = value; }
    }

    //Constructor
    public Human(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    //Methods
    public void WhoAmI(void)
    {
        Console.WriteLine("My name is " + name ".");
    }
}
```

We are now going to analyze this last declaration.

— Attributes : Lines 4 and 7 are attributes. These are fields that represent the object caracteristics. In this case, we can notice that the first one has the *public* prefix. This prefix means that the attribute will be accessible from outside the *class* bounds, and could be modified in that way. On the contrary, the second one has the *private* prefix. This means that the attribute will not be accessible from outside the *class scope*.

— Getter & Setter : the 10th declaration of a *get & set* property. In this example, the attribute age is not accessible from outside the *class scope*. Nevertheless, with the property *get* (*public* by default) of *Age* (Capital A to fit the convention), this attribute can be accessed from outside the *class* (only with the *get* property !). Similarly to *get*, the *set* property allows the user to change the value of an attribute.

— Constructor : line 17 is a declaration of a constructor. This function will create the object according to the class specifications. The prefix *this* is useful to specify which of the two variables belongs to the object itself. This means that we can name attributes and constructor parameters the same way, for more clarification. Be careful ! All the fields of the object instance must be assigned during the construction process.

— Methods : line 25 of the code snippet is a method definition. The methods are functions that belongs to the class in which they are declared. In this example, the method *WhoAmI()* can be executed on any *Human* type object.

2.4 Inheritance

Now we are going to see the concept of inheritance. This concept will be useful at least for the rest of the year.

```
public class Student : Human
{
    private ClassGroup grp;

    public Student(string name, int age, ClassGroup grp)
    {
        this.name = name;
        this.age = age;
        this.grp = grp;
    }
}
```

Let's get started by studying this code snippet. First, on line 1, we can recognize a class declaration, followed by the name of an existing class. This means the class declared *(here Student)* depends on another already declared class *(here Human)*. In that way, all declarations made in *Human class* will also be valid in this new class, **even if they do not appear in code.** Furthermore, new attributes and methods can be added, and will only belong to the *Student class.* **You must then remember to change the constructor consequently.**

Here, we have another example of inheritance :

```
public class Assistant : Human
{
    private ClassGroup grp;

    public Assistant(string name, int age, ClassGroup grp)
    : base(name, age)
    {
        this.grp = grp;
    }
}
```

The main difference between this class and the preceding one is that we do not bother to declare attributes from the *Human Class*. Indeed, the *base* prefix, followed by the corresponding arguments, permits the user to assign values to base class attributes automatically.

```csharp
public abstract class Human
{
    public Human(string name, int promotion)
    {
        this.name = name;
        year = promotion;
    }
}

//Not possible anymore
Human male = new Human("Jhon", 37);

//Valid declaration
Student major = new Student("Sarah", 19, ClassGroup.F1);
```

Now, let's say that we do not use the class *Human* anymore. This class has became obsolete : we won't create objects of its type. It is only here to generate classes that depends on it. We can then make this class abstract. An object from an *abstract* class **cannot be instantiated anymore.** Obviously, *abstract classes* are useful only if several other classes inherit from them. The code snippet above is an example.

2.5 Exceptions

Last but not least, the exception concept. This notion is important in oder to be able to handle impossible cases during implementation. Let's take the example of the factorial function, if the number is negative, it must not work. We are then going to see how to throw an exception, and how to catch them, in oder to avoid infinite programs. Triggering an exception is equivalent to exiting the program.

2.5.1 Throw an Exception

Here is how you will throw a new Exception from a specified type :

```csharp
1   public static void factor(int n)
2   {
3       if (n < 0)
4       {
5           throw new ArgumentException("N value must be positive");
6       }
7       if (n < 2)
8           return 1;
9       return n * factor(n - 1)
10  }
```

In this way, at line 5, we have thrown an *Exception* of type *ArgumentException*, if n is negative. The *throw new* prefix allows the user to throw a new *Exception* from specified type. This example is then followed by the message the exception must return.

2.5.2 Try Catch

We are now going to see how to catch an exception :

```
1  public static int launchFactor(int n)
2  {
3      try
4      {
5          return factor(n);
6      }
7      catch (Exception e)
8      {
9          Console.WriteLine("The number must be greater than 0");
10         Console.WriteLine(e.Message);
11         return 0;
12     }
13 }
```

The *try* and *catch* prefixes (line 3 and 7) permit us to catch an exception. In that way, the code in *try* part will be executed first, and if no error or exception is returned, the instructions after the *Try-Catch* will be executed. Nevertheless, if an exception from specified type (braces after *catch*) is triggered, the instructions in the *catch* part will be executed normally, along with the following ones. **Be aware ! If the exception does not match the type of *Exception* specified, the program will still exit without executing the other instructions.**

3 Exercises

3.1 Exercise 1

The aim of each threshold of this exercise is to learn of to throw Exceptions in impossible cases. **Please make sure to respect all function prototypes.**

3.1.1 Threshold 0

For this threshold, you must implement the function **Fibonacci**. This function will take as a parameter an *integer n* and will return the result. This function will trigger an *ArgumentException* if the n in entry is not valid.

```
public static int Fibonacci(int n)
{
    throw new NotImplementedException("FIX ME");
}
```

3.1.2 Threshold 1

For this threshold you will implement two convertors : One from Radians to Degrees (**RadToDeg**) and the inverse (**DegToRad**). However, these converors will only work if the *angle parameter* is included in the following interval :

$$[-\pi; \pi]$$

In every other case, an *ArgumentException* will be returned. Here are the prototypes to follow :

```
public static float DegToRad(float angle)
{
    throw new NotImplementedException("FIX ME");
}

public static float RadToDeg(float angle)
{
    throw new NotImplementedException("FIX ME");
}
```

3.1.3 Threshold 2

For this threshold, you will implement the function **Pow**. This function will take as a parameter a *float* representing the base number and an *integer* representing the power to apply. This function will return the result or an *OverflowException* if any overflow happens. **Be careful, you will need to handle, without exception, the case of negative powers**. Here is the prototype to follow :

```
public static float Pow(float n, int p)
{
    throw new NotImplementedException("FIX ME");
}
```

3.2 Exercise 2

In this exercise, you must implement a (simplified) Pokemon class and a Trainer class from the famous game Pokemon. To achieve this goal, you need to really understand the course part on classes, inheritance and enumerations, so reading the course again could be a good idea.

3.2.1 Threshold 0

For this threshold, you must implement a basic class, the Animal class. All your other classes will inherit from this one. This class must be implemented in the "Animal.cs" file.

```csharp
//Some usings

namespace miniPokemon
{
    public class Animal
    {
        public Animal(string name)
        {
            //FIXME
        }
    }
}
```

To be instantiated, a class needs a constructor that you must complete. The Animal class has a private attribute *name*, a public getter/setter *Name* and three public methods, emphWhoAmI(), *Describe()* and *Rename(string newName).*

```csharp
Animal cat = new Animal("Epicat");
cat.WhoAmI(); //Displays "I am an animal !"
cat.Describe(); //Displays "My name is Epicat."
cat.Name = "Moumoune"; //Doesn't work
cat.Rename("Moumoune"); //Changes the name
cat.Name; //return "Moumoune"
```

3.2.2 Threshold 1

For this threshold, you must implement a Pokemon class in the file "Pokemon.cs". This class inherits from the Animal class you've implemented. You can see this class as a general pokemon, it is neither a Pikachu or a Piplup, it's just a pokemon. The constructor of the Pokemon class is :

```csharp
public Pokemon(string name, int life, int damage, Poketype poketype)
:base(name)
{
    //FIXME
}
```

Here *poketype* represents a value of the enumeration below (you must include it in your Pokemon class) :

```
public enum Poketype
{
    POISON,
    FIRE,
    WATER,
    GRASS,
    ELECTRICK
};
```

The Pokemon class has several private attributes :

— a Poketype *poketype* which defines the type of a pokemon.
— an int *damage* which defines the attack damages of a pokemon.
— an int *level* which corresponds to the level of the pokemon. All pokemon starts with their level at 1.
— a boolean *isKO* which is true if the pokemon's life points are at 0.
— an int *life* which corresponds to the remaining life points of a pokemon.

Moreover, a Pokemon has different methods. The methods *WhoAmI()* and *Describe()*, inherited from Animal, should behave as follows :

```
Pokemon piplup = new Pokemon("Piplup", 100, 20, Pokemon.Poketype.WATER);
piplup.WhoAmI(); //Displays "I'm a Pokemon"
piplup.Describe(); //Displays "My name is Piplup I'm a pokemon
//of type WATER and I'm level 1"
```

A Pokemon has 4 public methods, *LevelUp()* which increments its level, *GetHurt(int damage)* which subtracts *damage* life points from the Pokemon, *Heal(int life)* which adds *life* life points to the Pokemon and finally *Attack()* which returns an int *damage*.

Plus, the class has a getter/setter for life, *Life*, and one for isKO, *IsKO*. Here is an example of the behavior of a Pokemon :

```
Pokemon  charmander = new Pokemon("Fire", 80, 15, Pokemon.Poketype.FIRE);
int lp = charmander.Life; //lp = 80;
int attack = charmander.Attack(); //Returns 15 and
//displays "Fire  uses cut, it's super effective"
charmander.GetHurt(90);
lp = charmander.Life; //lp = 0;
bool state = charmander.IsKO; //state = true;
charmander.Heal(80); //charmander.Life = 80 et salameche.IsKO = false
```

3.3 Threshold 2

For this threshold, you must implement the Trainer class which inherits from the Animal class. It must be implemented in the file "Trainer.cs". Now, you should be able to create the constructor of this class alone.

A trainer has a private attribute *age* (and a getter/setter *Age*), and a private list of Pokemon *listPokemon*. To implement this, you must use the List class. Use the MSDN page to understand how it works. Here is an example with int :

```
List<int> integer = new List<int>(); //initialisation of the list
integer.Add(2); //adds 2 to the list
integer.Add(3);
integer.Remove(2); //removes 2 from the list

//to display the content of a list

foreach(int i in integer)
{
    Console.WriteLine(i);
}
```

A trainer has different methods. *WhoAmI()* and *Describe()* should behave like this :

```
Trainer tom = new Trainer("Tom", 19);
tom.WhoAmI(); //Displays "I'm a pokemon Trainer !"
tom.Describe(); //Displays "My name is Tom, I'm 19
//and I have 0 Pokemon !"
```

A trainer has some other methods like *Birthday()* which increments its age, *NumberOfPokemon()* which returns the number of pokemon in the list, *CatchAPokemon(Pokemon pokemon)* whichs add a Pokemon to the list and finally *MyPokemon()* which displays all the pokemons of a trainer.

```
Trainer tom = new Trainer("Tom", 19);
tom.CatchAPokemon(new Pokemon("Pika", 50, 2, Pokemon.Poketype.ELECTRICK));
tom.CatchAPokemon(new Pokemon("Grimer", 90, 9, Pokemon.Poketype.POISON));
int count = tom.NumberOfPokemon(); //count = 2
tom.Birthday(); //tom.Age = 20
tom.MyPokemon();
```

The last line must display :
My Pokemon are :
- Pika
- Grimer

3.4 Bonus

This subject is short enough for you to have time to do some bonuses. Surprise us ! For example you could create different classes for different Pokemon as Charizard or Rattata (no, not Rattata...) and create a pokedex. Or you could use the methods *GetHurt()* and *Attack()* to create a fight system between Pokemon or even arenas. It's the moment to train yourself with these notions of classes which you'll use all year (and even beyond !).

These violent deadlines have violent ends.