# TP C#4 : Pokemon

## Consignes de rendu

A la fin de ce TP, vous devrez rendre une archive respectant l'architecture suivante :

```
rendu-tpcs4-prenom.nom.zip
|-- prenom.nom/
    |-- AUTHORS
    |-- README
    |-- Exceptions/
        |-- Exceptions.sln
        |-- Exceptions/
            |-- Tout sauf bin/ and obj/
    |-- miniPokemon/
        |-- miniPokemon.sln
        |-- miniPokemon/
                |-- Tout sauf bin/ and obj/
```

N'oubliez pas de vérifier les points suivants avant de rendre :

— Remplacez `prenom.nom` par votre propre login et n'oubliez pas le fichier `AUTHORS`.
— Les fichiers `AUTHORS` et `README` sont obligatoires.
— Pas de dossiers `bin` ou `obj` dans le projet.
— Respectez scrupuleusement les prototypes demandés.
— Retirez tous les tests de votre code.
— **Le code doit compiler !**

### AUTHORS

Ce fichier doit contenir une ligne formatée comme il suit : une étoile (*), un espace, votre login et un retour à la ligne. Voici un exemple (où `$` est un retour à la ligne et ␣ un espace) :

```
*␣prenom.nom$
```

Notez que le nom du fichier est `AUTHORS` sans extension. Pour créer simplement un fichier `AUTHORS` valide, vous pouvez taper la commande suivante dans un terminal :

```
echo "* prenom.nom" > AUTHORS
```

### README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les boni que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus).

# 1 Introduction

## 1.1 Objectifs

L'objectif de ce TP est de découvrir la programmation orientée objet (POO). Durant ce sujet vous allez découvrir des notions comme les Classes ou l'Héritage.
La POO est un paradigme : il s'agit d'une façon d'appréhender un problème algorithmique. Vous connaissez déjà l'approche fonctionnelle avec le CamL mais il en existe bien d'autres.
De plus, vous allez apprendre comment utiliser de Enumérations,des Enregistrements ou des Exceptions.

# 2 Cours

## 2.1 Enumérations

Les *énumérations* sont des types spéciaux qui permettent de représenter une liste de valeur. Ces valeurs possèdent toutes des noms différents dans le code et sont représentées sous forme d'entiers *(int)*. Il est donc possible d'assigner un entier spécial à une valeur voulue (comme ligne 8 dans la déclaration ci-dessous)

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

Nous pouvons voir ci-dessous comment utiliser les valeurs d'une *énumération*. La plus simple, et plus explicite, restant la première ligne. Les deux autres lignes permettent de vérifier que les valeurs d'une énumération sont bien représentées sous forme *d'entiers.*

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

## 2.2 Enregistrements

En C# une *structure (ou enregistrement)* est très similaire à une *classe*. Une *structure* peut contenir des variables et des méthodes. Il est préférable d'utiliser une *structure* à la place d'une *classe* si l'on souhaite s'en servir uniquement pour stocker des variables et ainsi représenter un groupement de variables. Pour définir une *structure* en C# :

```
1  public struct Point3D
2  {
3      double x;
4      double y;
5      double z;
6  }
```

Pour initialiser une *structure,* il suffit de faire :

```
1  Point3D p;
2  p.x = 10.0;   //Assignation du champ x
3  p.y = 5.5;    //Assignation du champ y
4  p.z = -2.75;  //Assignation du champ z
```

### 2.3 Classes

Pour comprendre comment fonctionnent les *classes*, rien de mieux qu'un exemple.

```
1  public class Human
2  {
3      //Public Attributes
4      public string name;
5
6      //Private Attributes
7      private int age;
8
9      //Getter & Setters
10     public int Age
11     {
12         get { return age; }
13         set { age = value; }
14     }
15
16     //Constructor
17     public Human(string name, int age)
18     {
19         this.name = name;
20         this.age = age;
21     }
22
23     //Methods
24     public void WhoAmI(void)
25     {
26         Console.WriteLine("My name is " + name ".");
27     }
28 }
```

Nous allons donc maintenant étudier cette déclaration de *classe.*

- Attributs : les lignes 4 et 7 sont des attributs. Ce sont des champs qui représentent les caractéristiques de l'objet que l'ont veut créer. Dans le cas présent on constate que le premier a le préfixe *public.* Ce préfixe signifie que cette attribut sera accessible en dehors de la *classe*, et peut donc être modifié de cette manière. Au contraire, le second attribut a le préfixe *private.* Ce préfixe signifie que l'attribut ne sera accessible que dans *le scope de la classe,* c'est-a-dire, que depuis *la classe elle-même.*
- Getter & Setter : la ligne 10 est la déclaration d'une propriété *get & set.* Dans cet exemple l'attribut *age* n'est pas visible en dehors de sa propre classe. Avec la propriété *get* (*public* par défaut) de *Age* (avec un A majuscule selon la convention) on peut accéder à la valeur de la variable (et seulement par cette propriété !) depuis l'extérieur. De la même manière

*set* change la valeur de l'attribut. Cela permet donc d'avoir des attributs non visibles depuis l'extérieur, mais tout de même modifiables.

— Constructeur : la ligne 17 est une déclaration du constructeur. Cette fonction va créer un objet selon les caractéristiques de la classe. Le mot *this* sert à lever l'indétermination lors d'une affectation d'un des champs de l'objet : *this* récupère le symbole appartenant a la classe. De ce fait, il est donc possible de nommer les variables du constructeur de la même façon que les attributs. **Attention ! Tous les champs de l'objet doivent être assignés lors de la création de l'instance.**

— Méthodes : la ligne 24 est la définition d'une méthode. Les méthodes sont des fonctions qui appartiennent a la classe dans laquelle elles sont déclarées. Dans l'exemple ci-dessus, *WhoAmI()* peut être exécutée par tout objet de type *Human*.

### 2.4 Héritage

Maintenant, nous allons aborder la notion d'héritage. Cette notion va vous servir tout le long du reste de l'année (voire même plus).

```
1  public class Student : Human
2  {
3      private ClassGroup grp;
4
5      public Student(string name, int age, ClassGroup grp)
6      {
7          this.name = name;
8          this.age = age;
9          this.grp = grp;
10     }
11 }
```

Étudions maintenant cette partie de code. Tout d'abord, à la ligne 1, on reconnaît une déclaration de classe, suivie par le nom d'une classe existante. Cela signifie que la classe déclarée *(ici Student)* dépends d'un autre classe déjà déclarée *(ici Human)*. De ce fait, toutes les déclarations faites dans la classe *Human* seront aussi valables, **même si elles n'apparaissent pas dans le code.** De plus des nouveaux attributs et des nouvelles méthodes peuvent être ajoutée, si ils ont un nom différents de ceux déjà présent. **Il faut donc penser à modifier le constructeur en conséquence.**

Ici, nous avons un autre exemple d'héritage :

```
1  public class Assistant : Human
2  {
3      private ClassGroup grp;
4
5      public Assistant(string name, int age, ClassGroup grp)
6      : base(name, age)
7      {
8          this.grp = grp;
9      }
10 }
```

La différence principale entre cette classe et le précédente, est qu'on ne s'embête pas à assigner les attributs déjà présents dans la *classe Human*. En effet, le préfixe *base* suivi des paramètres

nécessaires, permet d'assigner les valeurs aux attributs présents dans la classe de base de façon automatique.

```
1  public abstract class Human
2  {
3      public Human(string name, int promotion)
4      {
5          this.name = name;
6          year = promotion;
7      }
8  }
9
10 //Not possible anymore
11 Human male = new Human("Jhon", 37);
12
13 //Valid declaration
14 Student major = new Student("Sarah", 19, ClassGroup.F1);
```

Maintenant disons que nous n'utilisons plus la *classe Human.* Cette classe est donc devenue obsolète : nous ne créerons plus d'objets de ce type. *Human* n'est donc plus là que pour générer les classes qui héritent d'elle. Nous pouvons la rendre abstract. Un objet d'une classe *abstraite* **ne peut plus être instancié.** Bien évidemment les *classes abstract* n'ont d'intérêt que si plusieurs autres classes en héritent. Le code ci-dessus est un exemple.

## 2.5 Exceptions

Pour finir, nous allons aborder la notion d'exception. Cette notion est importante afin de gérer des cas d'implémentations impossibles. Prenons l'exemple de la fonction factorielle, si le nombre entre est négatif, cela ne doit pas fonctionner. Nous allons donc voir comment lancer une exception et comment les "attraper" pour éviter de faire tourner indéfiniment le programme. Lancer une exception, revient a quitter le programme.

### 2.5.1 Créer une Exception

Voici donc comment lancer une exception si n est négatif :

```
1  public static void factor(int n)
2  {
3      if (n < 0)
4      {
5          throw new ArgumentException("La valeur de n doit être positive");
6      }
7      if (n < 2)
8          return 1;
9      return n * factor(n - 1)
10 }
```

Ainsi, a la ligne 5 nous lancons une *Exception* du type *ArgumentException* si l'argument est négatif. Le préfixe *throw new* permet de lance une *Exception* du type spécifié ensuite. Cet exception est ici suivi du message qu'elle doit renvoyer.

### 2.5.2 Try Catch

Ici, nous allons donc voir comment "attraper" une 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 }
```

Les préfixes *try* et *catch* (ligne 3 et 7) permettent "d'attraper" une exception. Ainsi, le code dans la partie *try* sera exécutée normalement, et si tout se passe correctement les instructions après le *Try-Catch* seront exécutées. Cependant si une erreur/exception du type spécifié par le *catch* est lancée, le code dans la partie *catch* sera exécuté à la place, pour ensuite continuer sur les instructions suivantes sans interruption de programme. **Attention, si l'exception n'est pas incluse dans le type spécifié, le programme terminera sans exécuter le reste des instructions**.

# 3 Exercices

## 3.1 Exercice 1

Le but de cet exercice est d'apprendre à utiliser *les exceptions*. Dans chaque exercice, vous devrez lancer un type d'exception spécifié et dans le bon cas.

### 3.1.1 Palier 0

Pour le palier 0, vous devrez implémenter la fonction **Fibonacci**. Cette fonction prendra en paramètre un *entier n* et devra retourner un entier. Cette fonction retournera une *ArgumentException* si l'entrée est non valide.

```
1  public static int Fibonacci(int n)
2  {
3      throw new NotImplementedException("FIX ME");
4  }
```

### 3.1.2 Palier 1

Pour ce palier, vous aller réaliser deux convertisseur. Un Radian vers degré (**RadToDeg**) et l'autre l'inverse (**DegToRad**). Cependant, ce convertisseur ne prendra qu'en entrée des angles compris dans l'intervalle suivant :

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

Dans tous les autres cas, une *ArgumentException* sera retournée. Voici les prototypes à suivre :

```
1  public static float DegToRad(float angle)
2  {
3      throw new NotImplementedException("FIX ME");
4  }
5
6  public static float RadToDeg(float angle)
7  {
8      throw new NotImplementedException("FIX ME");
9  }
```

### 3.1.3 Palier 2

Pour ce palier, vous allez devoir implémenter la fonction **Pow**. Cette fonction prenant en paramètres un *float* représentant le nombre de base et un *entier* représentant la puissance. Vous devrez retourner le résultat de cette fonction, ou une *OverflowException* si un dépassement se produit. **Attention, il faudra gérer, sans exception, les cas où la puissance est négative.** Voici le prototype a suivre :

```
1  public static float Pow(float n, int p)
2  {
3      throw new NotImplementedException("FIX ME");
4  }
```

## 3.2 Exercice 2

Le but de cet exercice va être d'implémenter une classe *pokémon* (simplifiée) et une classe *dresseur*, du fameux jeu pokémon. Pour cela vous allez utiliser les notions de classes, d'héritages et d'énumérations, donc n'hésitez pas à relire une deuxième fois la cours pour être sur d'avoir compris.

### 3.2.1 Palier 0

Pour ce palier, vous allez commencer par implémenter une classe très basique, la classe *Animal*. C'est de celle ci qu'héritera toutes vos classes. Elle doit se trouver dans le fichier "Animal.cs".

```
1  //Des usings
2
3  namespace miniPokemon
4  {
5      public class Animal
6      {
7          public Animal(string name)
8          {
9              //FIXME
10         }
11     }
12 }
```

Pour pouvoir être instanciée, une classe a besoin d'un constructeur que vous devez compléter. La classe *animal* possède un attribut privé *name*, un getter public *Name* ainsi que trois méthodes publiques, *WhoAmI()*, *Describe()* et *Rename(string newName)*.

```
1  Animal cat = new Animal("Epicat");
2  cat.WhoAmI(); //Doit afficher "I am an animal !"
3  cat.Describe(); //Doit afficher "My name is Epicat."
4  cat.Name = "Moumoune"; //Ne doit pas marcher
5  cat.Rename("Moumoune"); //Doit changer le nom
6  cat.Name; //return "Moumoune"
```

### 3.2.2 Palier 1

Pour ce palier, vous allez implémenter la classe *Pokemon* dans le fichier "Pokemon.cs". Cette classe doit hériter de la classe *Animal* que vous avez codé précédemment. Vous pouvez voir cette nouvelle classe comme un pokémon universel, ce n'est ni un Pikachu, ni un Tiplouf, c'est juste un pokémon. Le constructeur de la classe *Pokemon* est le suivant :

```
1  public Pokemon(string name, int life, int damage, Poketype poketype)
2  :base(name)
3  {
4      //FIXME
5  }
```

Ici *poketype* représente une valeur dans l'énumération ci dessous (que vous devez inclure dans votre classe *Pokemon*) :

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

La classe *Pokemon* possède plusieurs attributs privés :

— un Poketype *poketype* qui définit le type du Pokémon.
— un int *damage* qui définit le nombre de dégâts infligés par une attaque.
— un int *level* qui correspond au level du Pokémon. Tous les pokémons commencent avec un level de 1.
— un booléen *isKO* qui est vrai si le Pokémon a ses points de vie a 0.
— un int *life* qui correspond aux points de vie du Pokemon.

En plus de ces attributs, un Pokémon a diverses méthodes. Les méthodes *WhoAmI()* et *Describe()*, héritées d'*Animal*, se comportent de la manière suivante :

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

Un Pokémon a 4 méthodes qui lui sont propres, *LevelUp()* qui incrémente le level, *GetHurt(int damage)* qui enlève *damage* points de vie au Pokémon, *Heal(int life)* qui lui en rajoute *life* et enfin *Attack()* qui retourne un int *damage*.

En plus de ça, la classe possède un getter/setter pour life, *Life*, et un pour isKO, *IsKO*. Voici un exemple du comportement d'un Pokemon :

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

### 3.3 Palier 2

Dans ce palier, vous allez implémenter la classe *Trainer* (dresseur en français), qui hérite elle aussi de la classe *Animal*. Elle devra se trouver dans le fichier "Trainer.cs". Normalement, vous devriez maîtriser les constructeurs maintenant.

Un dresseur a un attribut privé *age* (et un getter/setter *Age*) ainsi qu'une liste privée de Pokémons *listPokemon*. Pour cela vous allez utiliser la classe *List*. Nous vous conseillons d'aller consulter la page MSDN consacrée.

Cependant les listes s'utilisent de la manière suivante :

```
List<int> entier = new List<int>(); //initialisation de la liste
entier.Add(2); //ajout de l'int 2
entier.Add(3);
entier.Remove(2); //suppression de l'entier 2

//pour afficher le contenu d'une liste

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

Un dresseur a différentes méthodes. *WhoAmI()* et *Describe()* se comportent comme suit :

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

Un dresseur a aussi une méthode *Birthday()* qui incrémente son age, *NumberOfPokemon()* qui retourne le nombre de Pokémons dans la liste, *CatchAPokemon(Pokemon pokemon)* qui ajoute le pokémon à la liste et enfin *MyPokemon()* qui affiche tous les Pokémons d'un dresseur.

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

Cette dernière ligne doit afficher :
My Pokemon are :
- Pika
- Tadmorv

### 3.4 Bonus

Ce sujet est assez court pour que vous ayez le temps de faire des bonus. Surprenez nous, vous pourriez par exemple créer des classes pour différents Pokémons comme Dracaufeu ou Rattata (non pas Rattata...) et faire un pokédex. Ou alors avec les methodes *GetHurt()* et *Attack()* vous pourriez mettre en place des combats, voire même des arènes. C'est le moment de vous entrainer pour maitriser au maximum ces notions capitales !

**These violent deadlines have violent ends.**