TP C#7 : La Programmation Orientée Objets

1 Consignes de rendu

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

```
tpcs07-login_x.zip
    |– tpcs07-login_x/
        |– AUTHORS
        |– README
        |– src/
            |– tpcs07.sln
            |– tpcs07/
                |– gameplay/
                    |– Arena.cs
                    |– Attack.cs
                    |– Pokemon.cs
                |– Tout sauf bin/ et obj/
```

Vous devez, bien entendu, remplacer login_x par votre propre login. Avant de rendre, n'oubliez pas de vérifier :

- Que le fichier AUTHORS est bien au format habituel (une *, un espace, votre login et un retour à la ligne, dans un fichier sans extension)
- Que vous avez bien supprimé les dossiers bin et obj
- **Que votre code compile**

2 Cours

2.1 La Programmation Orientée Objet

La Programmation Orientée Objets(POO) permet de regrouper des données et des algorithmes dans une seule et unique entité : un objet. Chaque objet possède d'uniques attributs et méthodes, définis respectivement dans leur classe associée.

2.1.1 Classes et Objets

Une classe est un modèle de programmation, composée de la déclaration d'attributs et de méthodes. Une classe est représentée sous la forme :

```
1  class ClassName
2  {
3          /* Set of attributes */
4          int attribute1; // Attributes are
5          int attribute2; // simply variables
6
7          /* Set of functions */
8          void my_func1();
9          int my_func2();
10 }
```

Une classe contient donc des attributs et des méthodes qui lui sont liés. En revanche, une classe ne peut pas être utilisée telle quelle. Il faut, au préalable, instancier cette classe afin d'obtenir un objet. Bien sûr, une classe peut contenir des attributs qui sont eux même des objets.

Pour acceder aux informations propres à un objet, il suffit d'utiliser l'opérateur ".".

Exemple :

```
1  // Calls the print function contained in my_object
2  my_object.print();
3  // Set variable "age" of my_object to 12
```

```
4 my_object.age = 12;
5 // Prints variable "name" defined in my_object
6 Console.write(my_object.name);
```

En résumé, une classe n'est que la définition formelle d'un objet, qui lui est considéré par le compilateur comme un type à part entière, type dont on peut utiliser les méthodes et attributs associés. Chaque instance d'objet possède son propre espace mémoire. Par conséquent, les variables ne sont pas partagées entre les objets de même type. Si vous avez du mal à comprendre réellement ces deux notions, pas de panique, cela viendra au fil du TP et de la pratique.

2.1.2 Principe d'encapsulation

Lorsque l'on crée nos propres classes, il est possible de spécifier, devant nos attributs ou nos méthodes, les mots-clefs **private** ou **public**. Vous vous êtes surement déjà demandé à quoi ils correspondaient, eh bien, soyez heureux, c'est aujourd'hui que vous allez le découvrir ! =)

Ces deux mot-clefs permettent de spécifier la visibilité des méthodes et attributs de votre objet. Lorsque l'on spécifie nos champs en private, ceux-ci ne sont accessibles qu'au travers de notre objet, lors de la définition de la classe. En revanche, lorsque l'on spécifie le mot-clef **public**, les attributs et méthodes sont accessibles via l'exterieur de l'objet.

2.1.3 Instancier une classe : création d'un objet

A cet instant, une question devrait trotter dans votre tête. Mais comment que je le crée mon objet ? La syntaxe est assez simple, il suffit de faire :

```
1 TypeObjet mon_objet = new TypeObjet()
```

Ceci permet d'instancier un objet représenté par la classe **TypeObjet** et de stocker le résultat dans la variable **mon_objet**.

2.1.4 Exemple : Création d'une voiture

Prenons l'exemple d'un véhicule citadin motorisé de catégorie A, afin d'appliquer les notions vues précédemment. Les premières questions que vous devez vous poser sont :

— Un véhicule possède-t-il des attributs qui lui sont propres ? (taille, vitesse, nombre de roues, ...)
— Quels sont les actions (méthodes) qui permettent d'intéragir avec les attributs de votre véhicule ? (Déverouiller, accélérer, freiner, ...)

A partir de ces informations, nous pouvons élaborer notre classe, qui servira de modèle à l'instanciation de nos objets.

```
1  namespace Vehicle
2  {
3      class Vehicle
4      {
5          /* Set of attributes */
6          private string  model_;
7          private int     nb_wheels_;
8          private int     fuel_;
9          private float   max_speed_; // Max reachable speed
10         private float   speed_; // The current speed of the vehicle
11         private float   acceleration_; // The acceleration coefficient
12
13                 // Constructor used to initialize our attributes
14         public Vehicle(string model, int nb_wheels, int fuel,
15                                 float max_speed, float acceleration)
16         {
17             this.model_ = model;
18             this.nb_wheels_ = nb_wheels;
19             this.fuel_ = fuel;
20             this.max_speed_ = max_speed;
21             this.acceleration_ = acceleration;
22
23             this.speed_ = 0.0f; // The vehicle is stopped by default
24         }
25
26         // Increases the speed and consumes one unit of fuel
27         public void accelerate(float friction)
28         {
29             speed_ += friction * acceleration_;
30             fuel_--;
31         }
32
33         // Decreases the speed and clamp it to 0
34         public void brake(float friction)
35         {
36             speed_ -= friction * acceleration_;
37             if (speed_ <= 0.0f)
38                 speed_ = 0.0f; // Clamp the speed to 0 km/h
39         }
40
41         public void fill(int amount)
```

```
42          {
43              fuel_ += amount;
44          }
45
46          public void unlock()
47          {
48              Console.WriteLine("* Bip Bip *");
49          }
50
51          /* Getters */
52          public string model_get()
53          {
54              return model_;
55          }
56
57          public int nb_wheels_get()
58          {
59              return nb_wheels_;
60          }
61
62          public float speed_get()
63          {
64              return speed_;
65          }
66
67          public bool is_empty()
68          {
69              return fuel_ == 0;
70          }
71      }
72  }
```

Comme vous pouvez le voir, tous nos attributs sont **private**. Nous ne pouvons donc modifier leur valeur uniquement via la classe **Vehicle**. Il est, par exemple, impossible d'effectuer ceci :

```
1  Vehicle my_car = new Vehicle("Catrel", 100, 20, 1.1f);
2  Console.Write(my_car.model_); // Impossible : model_ is private
```

3 Exercices

Au fil de ce TP, vous allez mettre en place un système de combat Pokémon. Comme d'habitude, les fonctions sont données, il ne vous reste plus qu'à les remplir.

3.1 Information

Afin de tester votre TP, vous devez vous rendre dans le fichier **Program.cs**, instancier deux pokémons, créer une nouvelle arène à partir de ces deux instances et de décommenter la ligne suivate :

```
1 //Application.Run(new Form1(arena));
```

en passant au constructeur de **Form1** votre instance d'arène.

3.2 Exercice 1 : Class Attack

Dans cet exercice, vous devez remplir la classe **Attack.cs**, qui sera utilisée par nos pokémons pour s'attaquer entre eux.

3.2.1 Constructeur

Complétez le constructeur de la classe Attack dans le fichier **Attack.cs**. Celui-ci se charge uniquement d'initialiser les différents attributs.

Prototype :

```
1 public Attack(string name, int min, int max, int crit_rate,
2                int crit_fail_rate, int crit_bonus_rate)
3 {
4         // FIXME
5 }
```

3.2.2 La puissance de nos attaques

Complétez la fonction ci-dessous, qui renvoie un nombre de dommages en fonction d'un min, d'un max et d'un taux de criticité (oui, ça existe). Votre fonction doit utiliser un nombre aléatoire et les points donnés sur cet exercice dépendront en grande partie de l'implémentation trouvée.

Prototype :

```
1 public int damage_get()
2 {
3         // FIXME
4 }
```

3.2.3 Récupérer le nom d'une attaque

Complétez le getter **name** de la classe **Attack**. Attention, l'accès à name ne doit pas permettre de changer la valeur du nom.

Prototype :

```
1 public string name { // FIXME }
```

3.3 Exercice 2 : Mon premier Pokemon !

Vous commencez à être à l'aise avec le fonctionnement des constructeurs. C'est pourquoi nous avons décidé de vous laisser face à vous même en ce qui concerne l'implémentation des constructeurs de classes que nous verrons au cours de ce TP.

3.3.1 Obtenir les informations de notre pokémon

Complétez la fonction **who_am_i**, décrite par le prototype ci-dessous. Dans cette fonction, vous devez afficher le nom du pokémon, suivi de :

— "and I am quite weak" si son niveau est inférieur à 10

— "and I am still under training" si son niveau est inférieur à 25
— "and I am ready to battle" si son niveau est inférieur à 50
— "and I am quite strong" si son niveau est inférieur à 75
— "and you cannot beat me" dans tous les autres cas

Prototype :

```
1 public string who_am_i()
2 {
3         //FIXME
4 }
```

3.3.2 Appliquer une attaque sur un Pokémon

Complétez la fonction **undergo**, qui se charge d'appliquer les dommages d'une attaque sur un Pokémon.

Prototype :

```
1 public void undergo(Attack a)
2 {
3         //FIXME
4 }
```

3.3.3 Getters/Setters

Pour valider cet exercice, vous devez compléter les Getters/Setters suivants :
— **name** : retourne le nom du Pokémon.
— **type** : retourne le type de Pokémon.
— **pv** : retourne le nombre de points de vie du Pokémon.
— **pv_max** : retourne le nombre de point de vie maximum du Pokémon.
— **lvl** : retourne le level du Pokémon.
— **set_attack** : ajoute une attaque à la position n du tableau (le premier index étant 0).
— **is_alive** : retourne vrai si le pokémon a encore de la vie, faux autrement.

3.4 Exercice 3 : L'heure de la bataille a sonné

Bien ! Nous avons maintenant des attaques et des pokémons. Seul problème, il faut maintenant les emmener vers le champ de bataille, afin de profiter du spectacle, sadiques que nous sommes. Dans cet exercice, vous devez compléter la classe **Arena** disponible dans le fichier **Arena.cs**

3.4.1 Le tour par tour

Vous devez compléter la fonction change_attaker, qui permet d'échanger les pokémons représentés par **undergoing_** et **attacking_**.

Prototype :

```
1 public void change_attacker()
2 {
3         //FIXME
4 }
```

3.4.2 Blesser l'ennemi pour mieux gagner

Vous devez compléter la fonction **attack_with**, qui se charge de blesser le pokémon qui n'est pas attaquant, **n** représentant l'indice de l'attaque dans le tableau d'attaque.

Prototype :

```
1 public void attack_with(int n)
2 {
3         //FIXME
4 }
```

3.4.3 C'est bon, on a fini ?

Vous devez compléter la fonction is_finished, qui indique si le combat entre les deux pokémons est terminé ou non. Prototype :

```
1 public void is_finished()
2 {
3         //FIXME
4 }
```

3.4.4 Getters/Setters... toujours plus !

Vous devez compléter les deux getters/setters suivants :

— **left** : Renvoie / Modifie la valeur pointée par la variable **mine_**

— **right** : Renvoie / Modifie la valeur pointée par la variable **opponent_**

Afin de tester votre tp, nous vous rappelons que vous devez vous rendre dans le fichier Program.cs, instancier deux pokémons afin de créer une nouvelle arène, décommenter la ligne suivante :

```
1 //Application.Run(new Form1(arena));
```

en passant au constructeur de Form1 votre instance d'arène.

3.5 Exercice 4 : Des bonus, toujours plus de bonus

3.5.1 Jouer des sons à chaque attaque

Pour valider ce bonus, vous devez ajouter un son pour l'ensemble des attaques existantes. Et pourquoi pas un peu de musique ? Vous pouvez ajouter de nouveaux pokémons, des effets... Impressionnez-nous !

3.5.2 Ajout d'un menu

Bon, c'est bien beau d'instancier des pokémons à la main, dans le code, mais nous, on voudrait pouvoir choisir notre pokémon pendant le jeu bon sang ! Vous pouvez donc ajouter un bonus, qui permet de sélectionner les deux pokémons et de leur ajouter des attaques, avant de démarrer le tant attendu combat.

3.5.3 Encore une fois... Impressionnez-nous !

Vous pouvez ajouter autant de bonus que vous le souhaitez (bon, faites attention tout de même, nous sommes en ING1, il faut nous ménager). Tous les bonus seront pris en compte, n'oubliez pas de les ajouter à votre README pour faciliter la correction.