TPC# 5

1 Consignes de rendu

Vous devrez soumettre une archive respectant l'architecture suivante :

```
rendu-tp-prenom.nom.zip
        |-- rendu-tpcs5-prenom.nom/
                |-- AUTHORS
                |-- README
                |-- Explorer/
                        |-- Explorer.sln
                        |-- Explorer/
                                |-- Tout sauf bin/ et obj/
```

Bien entendu, vous devez remplacer *prenom.nom* par votre propre login. N'oubliez pas de vérifier les points suivants avant de rendre votre partiel :

- Le fichier AUTHORS doit être au format habituel : ** prenom.nom$* où le caractère '$' **représente un retour à la ligne. Vous ne devez pas écrire de $ en dur dans votre fichier AUTHORS.**
- Le fichier README doit contenir les difficultés que vous avez rencontrées sur ce sujet, et indiquera les bonus que vous avez réalisés. **Il ne doit pas être vide**.
- Vous devez suivre le sujet **scrupuleusement**, et notamment **respecter les prototypes** des différentes fonctions.
- **Pas de dossiers bin et/ou obj dans le rendu final.**
- **Votre code doit compiler !**

2 Avant de commencer

Pour ce TP vous allez découvrir beaucoup de nouvelles notions qui ne sont pas toujours faciles à appréhender. Vous trouverez sûrement ce sujet plus difficile que les précédents : **n'hésitez pas à demander de l'aide à vos assistants et assurez-vous de lire et de comprendre la partie cours.**

3 Cours

Vous allez découvrir aujourd'hui la programmation orientée objet en C#. Il s'agit d'un *paradigme* de programmation, c'est à dire une représentation/un modèle du monde : en d'autre termes cela affecte la manière d'exprimer des idées et la manière de penser. (Le fonctionnel, que vous avez aperçu avec le caml est un autre paradigme).

3.1 Les Classes

Les classes sont une manière de représenter un type de données grâce à des propriétés. L'une des idées derrière la notion de classe est de regrouper et de représenter de la même manière des objets/concepts possédant des propriétés communes. On peut créer des instances de classe que l'on appellera **Objets**.

Les propriétés des classes peuvent se décomposer en 2 grands groupes :
- Les Attributs, il s'agit de données.
- Les Méthodes, qui définissent un comportement.

Les Attributs. Il s'agit de variables déclarées dans la classe. il existe deux types d'attributs : les attributs statiques qui sont stockés dans la classe et ont la même valeur pour toutes les instances de la classe. Le deuxième type d'attributs sont les attributs dits d'*instance*, ils ont une valeur différente par objet et sont stockés dans ces derniers.

Les Méthodes. Il s'agit de fonctions/procédures définies dans la classe, elles sont communes à toutes les instances de la classe et elles sont liées aux objets (le résultat dépendra des attributs de l'instance) : Elles nécessitent une instance pour être appelées. De la même manière que pour les attributs, il existe des méthodes statiques, ces dernières ne sont pas liées à un objet mais à la classe et ne nécessitent pas d'objets pour être appelées. C'est le cas de la méthode suivante que vous connaissez bien :

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

En effet, ici Console est une classe et WriteLine est une méthode statique.

Exemple : Déclaration d'une classe.

```
1  public class MyClass
2  {
3          //you can declare attributes of any type
4          //(even one define by another class of yours)
5          public static string myStaticAttribute = "A random value";
6          public int myAttribute;
7
8          public static string myStaticMethod()
9          {
10                 //here the instructions
11                 //for exemple return myStaticAttribute
12                 return myStaticAttribute;
13         }
14         //of course the return type can vary
15         public void myMethod()
16         {
17                 //here the instructions
18                 //for exemple write myAttribute's value:
19                 Console.WriteLine(myAttribute);
20         }
21
22 }
```

3.2 Les Objets

Le processus de création d'un objet à partir d'une classe s'appelle l'**instanciation**. Une classe permet d'instancier des objets similaires.

Les Constructeurs : Un *constructeur* est une procédure spéciale qui permet de créer/construire un objet instance d'une classe. Un constructeur vide est fourni par défaut mais vous pouvez créer votre propre constructeur et lui faire prendre les paramètres et effectuer les instructions que vous souhaitez pour initialiser vos objets.

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

Exemples d'instanciation :

```
1 /*
2 remarquez, même si le constructeur utilisé ici (celui par defaut)
3 ne prend pas de paramètres, on met des parenthèses car il s'agit d'un
4 appel à une méthode spéciale.
5 */
6 MyClass myObject = new MyClass();
```

Le constructeur par défaut ne fait rien, les attributs de mon objet ne seront donc pas initialisés et une erreur aurait lieu si je tentais d'y accéder. Pour y remédier, je peux définir mon propre constructeur **à l'intérieur de ma classe :**

```
1 //Notice how the constructor should have the name of the class
2 public MyClass(int attr)
3 {
4         //I can initialise my attributes the way I want here
5         myAttribute = attr;
6 }
```

Une fois que mon objet est créé (et que ses attributs sont initialisés, je peux y accéder de la manière suivante :

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

Il est possible d'accéder aux attributs et méthodes statiques sans créer d'objets :

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

3.3 Accessibilité

Il est possible de limiter l'accès à certaines propriétés de votre classe grâce aux *access modifiers* :

- **public** : les propriétés *publiques* sont accessibles depuis l'extérieur de votre classe (par l'intermédiaire d'un objet si il s'agit de propriétés d'instance).
- **private** : les propriétés *privées* sont accessibles depuis l'intérieur de votre classe uniquement et ne sont pas transmises aux classes filles en cas d'héritage.
- **protected** : les propriétés *protégées* sont accessibles depuis l'intérieur de votre classe uniquement mais sont transmises aux classe filles qui héritent de votre classe.

Par défaut, dans une classe, les attributs et méthodes sont privés, si vous souhaitez y accéder depuis l'extérieur de votre classe (via une instance/objet) vous pouvez soit leur donner la visibilité *public* soit **pour vos attributs** créer des méthodes qui vous serviront d'intermédiaires pour accéder et modifier ces attributs, on appelle ces méthodes des accesseurs. Bien entendu, les accesseurs doivent être publiques.

3.4 L'héritage

L'héritage est l'une des notions clés de la programmation orientée objet. Cela permet de regrouper des attributs et méthodes communes à plusieurs classes. Pour ce faire on définit une classe qui contient ces propriétés, il s'agit de la classe de base ou classe mère. Les classes qui possèdent ces attributs hériteront alors de la classe mère et définiront en plus des attributs et méthodes qui leur sont propres :

```
1  public class Vehicle
2  {
3          public string brand;
4          public int maxSpeed;
5  }
6  public class Car : Vehicle
7  {
8          //define here your 'Car' class
9  }
10 public class Bike : Vehicle
11 {
12         //define here your 'Bike' class
13 }
```

3.5 Les Structures

Les structures constituent une autre manière de définir un type de données. Les structures sont très similaires aux classes. Pour déclarer une structure, utilisez le mot-clé *struct* :

```
1  public struct Point
2  {
3          int x;
4          int y;
5  }
```

Pour initialiser une variable du type de votre structure, vous pouvez la déclarer de la même manière qu'une variable habituelle :

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

Par défaut, contrairement aux classes, les structures on une visibilité publique, ce qui vous permet d'accéder aux membres depuis l'extérieur de la structure.

3.6 Les Enumérations

Les énumérations constituent une manière très pratique de définir plusieurs constantes. Pour déclarer une énumération, utilisez le mot-clé *enum* :

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

Vous pourrez ensuite déclarer des variables du type de votre énumération :

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

3.7 Les Exceptions

Les exceptions sont *levées* lorsqu'une instruction ne peut pas être exécutée. Vous en avez sûrement déjà rencontrées plusieurs :

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

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

Pour contrôler l'exécution de votre programme et éviter les erreurs dues aux exceptions, vous pouvez/devez utiliser la structure du **try-catch** :

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

Le mot-clé **catch** et le bloc délimité par les accolades qui le suivent permettent de définir les instructions à exécuter en cas d'exception.

Les exceptions sont des objets de la classe *Exception*, vous pouvez donc définir votre propre type d'exceptions.

4 Pratique : Explorer

Vous allez maintenant pouvoir mettre en pratique les notions expliquées dans la partie cours : Le but de ce TP est de réaliser un jeu tout simple dans la console.

4.1 Les règles du jeu

Le jeu se déroule dans une grille en deux dimensions de longueur et largeur finies. Vous contrôlerez un personnage et votre but sera de terminer le niveau que représente la grille : dans un premier temps il s'agira d'atteindre un objectif situé quelque part sur la grille. Ensuite, vous ajouterez des ennemis et vous devrez les éliminer pour terminer le jeu.

Ce jeu se déroulera en "tours" : à chaque tour, vous effectuerez une action : dans un premier temps il ne vous sera possible que de bouger. Ensuite vous pourrez également attaquer un ennemi si celui-ci se trouve dans une case voisine (pas en diagonale).

Lorsque vous aurez ajouté les ennemis à votre jeu, votre personnage perdra de la vie à chaque tour qu'il termine à côté d'un ennemi. Vous perdez si vos points de vie tombent à 0.

4.2 Partie 1 : La Carte/Grille

Créez une classe *Map* pour représenter la carte de votre jeu : pour la première étape vous n'aurez besoin que de 4 attributs : un tableau d'entier à 2 dimensions (ainsi que les dimensions) et un Personnage :

```
1 public class Map
2 {
3         char[,] map;
4         int width;
5         int height;
6         Character hero;
7         //int obj_x;
8         //int obj_y;
9 }
```

Constructeur Ajoutez ensuite un constructeur à votre classe. Vous devez être capable de créer des cartes de différentes tailles, ajustez les arguments de votre constructeur en conséquence.

```
1 public Map(/*put here arguments for the constructor*/)
2 {
3         //initialize your class' attributes
4 }
```

Méthodes Votre classe *Map* devra contenir au moins 3 méthodes :

- *display()* pour afficher votre carte après chaque tour.
- *update()* pour mettre à jour votre carte à chaque tour.
- *is_ over()* pour détecter si le joueur a terminé le niveau (soit en atteignant l'objectif soit en ayant éliminé tous ses ennemis).

Pour avoir un affichage qui s'actualise et ne pas afficher la nouvelle version de la carte en dessous de la précédente, vous pouvez utiliser la méthode *Console.Clear()* pour nettoyer votre console avant d'afficher votre carte.

Faites bien attention à la visibilité de vos attributs et méthodes

4.3 Partie 2 : Le Joueur

Vous allez maintenant devoir créer une classe *Character* pour représenter le joueur, ou plus précisément son personnage. Le but ici est de permettre à l'utilisateur d'interagir avec son personnage pour le déplacer sur la carte et atteindre un objectif. Votre personnage doit posséder au moins les attributs suivants :

```
1 public class Character
2 {
3         int life;//store the character HPs
4         int x;//position in the x axis
5         int y;//position in the y axis
6 }
```

Constructeur Ici encore, votre but sera de créer un constructeur pour votre classe. Le bon choix peut être de créer un constructeur qui vous permet de choisir les points de vie de votre personnage pour augmenter la difficulté lorsque vous ajouterez les ennemis.

Pour l'instant vous aurez besoin d'implémenter uniquement la méthode *move* qui vous permettra de déplacer votre personnage.

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

Attention, ce n'est pas dans cette méthode que vous demandez un *input* pour déterminer le déplacement à effectuer. Ici, vous vous contenterez de mettre à jour les attributs de votre classe.

Là encore, faites bien attention à la visibilité de vos attributs et méthodes

4.4 Partie 3 : la fonction main

Une fois que vous avez créé vos classes *Map* et *Character*, il serait bien de les faire fonctionner ensemble et de jouer à votre jeu.

Votre fonction *main* devrait prendre la forme d'une boucle pseudo-infinie qui s'arrête lorsque le joueur à terminé le niveau, c'est à dire atteint l'objectif. A chaque itération de votre boucle, vous devrez mettre à jour la carte et l'afficher.

Il sera ensuite question de détecter les actions de l'utilisateur (les touches du clavier) pour déplacer le personnage :

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

Notez bien que cette fonction est bloquante : elle va stopper l'exécution de votre programme tant que vous n'avez appuyé sur aucune touche.

4.5 Partie 4 : Les ennemis

La dernière étape est d'ajouter des ennemis qu'il vous faudra éviter ou éliminer pour terminer le jeu. Pour les éliminer, il vous faudra être à côté d'un ennemi et d'appuyer sur la touche entrée (*ConsoleKey.Enter*). Pour cela, il vous faudra vérifier les voisins du personnage en parcourant la grille.

Les Ennemis Pour ajouter les ennemis, vous allez pouvez ajouter une classe *Ennemy*. Une méthode serait de faire hériter cette classe de votre *Character* et d'y ajouter des méthodes propres aux ennemis.

4.6 Conclusion

Ce tp n'est pas facile et vous êtes laissé assez libre dans la réalisation de votre jeu. Assurez vous de bien comprendre les notions de cours, elles vous seront très utiles par la suite.

Vous avez la possibilité de rajouter énormément de bonus, profitez en pour épater vos assistants !

"The Code is the Law"