TP C#12

1 Consignes de rendu

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

```
rendu-tp-prenom.nom.zip
    |-- rendu-tp-prenom.nom/
        |-- AUTHORS
        |-- README
        |-- Matrix/
            |-- Matrix.sln
            |-- Matrix/
                |-- Tout sauf bin/ et obj/
        |-- Tron/
            |-- Tron.sln
            |-- Tron/
                |-- 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 TP :

- Le fichier AUTHORS doit être au format habituel : ** prenom.nom$* où le caractère '$' représente un retour à la ligne.
- Vous devez suivre le sujet **scrupuleusement**, et notamment **respecter les prototypes** des différentes méthodes.
- Pas de dossiers bin et/ou obj dans le projet.
- **Votre code doit compiler !**

2 Introduction

2.1 Objectifs

Au cours de ce TP, nous aborderons les notions suivantes :

- Surcharge d'opérateurs
- Généricité

3 Cours

3.1 La surcharge

Vous avez normalement déjà vu et utilisé la surcharge lors de vos différents TP mais nous allons faire une petite révision. Le concept de surcharge (ou overloading) des méthodes est très simple. Il permet d'avoir plusieurs méthodes ayant le même nom mais prenant des arguments différents. Nous pouvons prendre comme exemple la méthode `MiniCat` du TPCS6 :

```
1 static void MiniCat(string InputFile);
2 static void MiniCat(string InputFile, string OutputFile);
```

ou même plus simplement une fonction que vous utilisez tous les jours dans la classe `Console:`

```
1 public static void WriteLine(bool value);
2 public static void WriteLine(char value);
3 public static void WriteLine(double value);
4 public static void WriteLine(int value);
```

3.1.1 Surcharge des opérateurs

En y réfléchissant bien on se rend compte que les opérateurs comme le + ou le - sont des méthodes. On y a juste ajouté du sucre syntaxique afin de permettre une écriture infixe comme en mathématiques. En effet, en Lisp par exemple, la notation infixe n'est pas native, pour effectuer une addition on doit donc écrire `+ x y` pour faire `x + y`. On voit donc bien ici que c'est la fonction `+` appliquée aux arguments x et y. Maintenant que l'on sait que les opérateurs sont de simples méthodes, pourquoi ne pas les surcharger ?

Prenons un exemple où l'on souhaite créer un objet `Vector`, et l'on veut pouvoir soustraire 2 vecteurs. On pourrait très bien faire une méthode `VectorSubtract` qui prendrait 2 vecteurs en argument et aurait un vecteur en valeur de retour, mais ce n'est pas assez intuitif, on veut pouvoir faire `vectorFinal = vector1 - vector2`. Cependant le C# ne peut pas le faire automatiquement, il va falloir lui expliquer comment faire, et pour cela il suffit de surcharger l'opérateur - grâce au mot clé `operator`.

Voyons un exemple avec notre classe `Vector` :

```
1  public class Vector
2  {
3      int x;
4      int y;
5      int z;
6
7      public Vector()
8      {
9          this.x = 0;
10         this.y = 0;
11         this.z = 0;
12     }
13
14     public Vector(int x, int y, int z)
15     {
16         this.x = x;
17         this.y = y;
18         this.z = z;
19     }
20
21     public static Vector operator -(Vector v1, Vector v2)
22     {
23         Vector result = new Vector();
24         result.x = v1.x - v2.x;
25         result.y = v1.y - v2.y;
26         result.z = v1.z - v2.z;
27         return result;
28     }
29 }
```

Vous remarquerez que le constructeur est surchargé et possède donc 2 variantes, on peut donc maintenant instancier un vecteur de 2 manières différentes : soit vide (coordonnées initialisés à 0), soit avec une valeur donnée pour chaque coordonnée.

Nous pouvons aussi remarquer qu'avec la surcharge de l'opérateur nous avons juste maintenant à lancer `vector1 - vector2` afin de réaliser une soustraction de deux vecteurs.

3.2 Les génériques, c'est pas automatique

Pour ceux ayant déjà fait un peu de C++, la notion de `template` ne doit pas vous être inconnue, elle existe aussi en C# sous le nom de `generics`. Note : Pour les curieux, les `generics` du C# sont moins flexibles que les `templates`, la liste des différences entre les `templates` en C++ et les `generics` en C# peut être trouvée ici : `https://msdn.microsoft.com/en-us/library/c6cyy67b.aspx`

3.2.1 Explication

Le but principal des `generics` est de traiter de manière identique des types ou structures de données différentes. Vous avez déjà utilisé cette notion sans le savoir, en effet les listes utilisent les `generics` : vous pouvez créer une liste d'`int` ou de `string` et les utiliser de manière totalement similaire en indiquant simplement le type de la liste entre `<>` lors de son instanciation.
Exemple :

```
1 List<int> l1 = new List<int>()
2 List<string> l2 = new List<string>()
```

3.2.2 Exemple

Reprenons notre exemple du vecteur, mettons que l'on veuille aussi pouvoir créer un vecteur où les coordonnées sont des flottants. Deux solutions :

— Créer une deuxième classe que l'on appellera `Vector_float`, copier-coller [1] notre code précédent dedans et changer tous les `int` par des `float` de manière (très) sale.
— Modifier notre classe en utilisant les `generics`.

Nous allons bien évidemment prendre la deuxième solution, pour cela nous aurons besoin de spécifier que l'on utilise un vecteur du type général `T` et non plus un vecteur d'entiers(`T` est une convention vous pouvez l'appeler comme bon vous semble), il suffira ensuite de remplacer toutes nos occurrences de `int` par des `T`.

1. copier-coller, c'est mal !

Exemple :

```
public class Vector<T>
{
    T x;
    T y;
    T z;

    public Vector()
    {}

    public Vector(T x, T y, T z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public static Vector<T> operator -(Vector<T> v1, Vector<T> v2)
    {
        Vector<T> result = new Vector<T>();
        result.x = (dynamic)v1.x - (dynamic)v2.x;
        result.y = (dynamic)v1.y - (dynamic)v2.y;
        result.z = (dynamic)v1.z - (dynamic)v2.z;
        return result;
    }

    public override String ToString()
    {
        return x + "," + y + "," + z;
    }
}
```

Comme vous pouvez le voir, nous n'avons pas eu à changer grand chose, nous avons ajouté `<T>` à `Vector` à chaque fois qu'il désignait un type (donc partout sauf pour les constructeurs), et nous avons remplacé tous les `int` par `T`, rien de plus.

3.2.3 Rien de plus ?

Les plus attentifs d'entre vous auront remarqué l'apparition du mot clé `dynamic` lors de la soustraction des valeurs du vecteur. En effet, sans ce mot clé, monodevelop refuse de compiler nous indiquant que l'opérateur - ne peut pas être appliqué aux opérandes de type `T` et `T`. Donc si vous avez bien compris la partie précédente sur la surcharge d'opérateur, monodevelop nous dit : je ne trouve pas de surcharge d'opérateur - pour le type `T` (plus simplement encore pour ceux qui seraient perdus : je ne sais pas soustraire deux objets de type `T`).

En effet c'est logique, `T` peut représenter n'importe quoi, cela peut très bien être un objet voiture et dans ce cas là il n'y a aucun sens à soustraire 2 voitures ! Donc la solution serait de surcharger `T` sur la soustraction ? Oui, mais comment ? Vu que `T` est général nous n'avons aucune idée de comment effectuer une soustraction. La vraie réponse est qu'on ne peut pas savoir de quel type sera `T` à la compilation, et on va donc dire explicitement au compilateur avec le mot clé `dynamic` que l'on veut que la vérification de la possibilité de soustraire 2 objets de types `T` soit faite à l'exécution et non à la compilation. En effet, lors de l'exécution `T` est remplacé par son vrai type et l'on peut donc savoir si l'on soustrait des `int` ou des voitures.
ATTENTION cela n'enlève pas complètement le problème. En effet, si le type `T` n'a toujours pas de surcharge sur l'opérateur -, il y aura une erreur lors de l'exécution. Cela peut très bien être gérée en C# grâce à l'ajout de contrainte sur le type en utilisant le mot clé `where`. Nous n'en parlerons pas lors de ce TP mais si cela vous intéresse, vous pouvez récupérer plus d'informations sur ce lien : `https://msdn.microsoft.com/en-us/library/bb384067.aspx`

3.2.4 Et c'est pas fini !

Dernière petite info de ce TP, dans notre classe `Vector` nous avons une méthode `ToString()`. Il faut savoir que n'importe quelle classe en C# hérite implicitement de la classe `Object`. Cette classe contient des méthodes qui pourront être remplacées (grâce au mot clé `override`) par toutes les autres classes. Nous avons pour exemple ici la méthode `ToString()` qui renvoie une string représentant notre classe (qui pourra donc être affichée). Pour plus d'info vous pouvez visiter ce site :
`https://msdn.microsoft.com/en-us/library/system.object(v=vs.110).aspx`

4 Exercices

4.1 Enter the Matrix

Cette partie est à coder dans le projet `Matrix`.
Commençons par un petit échauffement avec l'implémentation des matrices en C#. Nous parlons ici de la définition mathématique des matrices.

Pour faire cela, vous allez tout d'abord créer un classe générique `Matrix` qui aura comme seul attribut un tableau à double dimension.

4.1.1 Constructeur

Tout d'abord afin que notre classe puisse fonctionner, il lui faut un constructeur. Nous trouvons qu'un seul constructeur c'est faible, vous allez donc en coder trois !

```
1 public Matrix(int dim);
2 public Matrix(int height, int width);
3 public Matrix(int height, int width, T init);
```

Le premier va utiliser `dim` afin de créer une matrice carrée et ne pas assigner de valeur dans le tableau.
Le deuxième, lui, va utiliser les deux paramètres afin de donner une taille au tableau mais sans y insérer de valeur.
Le troisième, lui, va faire la même chose que le deuxième mais en initialisant toutes les cases du tableau avec la valeur `init`.
Il faudra lancer des `ArgumentException` avec un message approprié si les dimensions sont négatives.

4.1.2 Addition/Soustraction

Maintenant que l'échauffement de l'échauffement est terminé nous pouvons commencer les opérations sur nos matrices.

Vous allez donc devoir implémenter l'addition/soustraction matricielle. Bien sûr, vous devez surcharger l'opérateur + et -. Vous avez donc à implémenter les méthodes suivantes :

```
1 public static Matrix<T> operator -(Matrix<T> a, Matrix<T> b);
2 public static Matrix<T> operator +(Matrix<T> a, Matrix<T> b);
```

Bien sûr vous devez vérifier les dimensions des matrices et envoyer une `ArgumentException` en cas d'incompatibilité (hauteur ou largeur différentes).

4.1.3 Multiplication

Vous allez devoir maintenant implémenter la multiplication matricielle :

```
1 public static Matrix<T> operator *(Matrix<T> a, Matrix<T> b);
```

Bon comme on est gentil, on vous offre la formule de la multiplication :

$$\forall i, j : c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} b_{nj}$$

Comme pour l'addition/soustraction vous devez vérifier les dimensions et envoyer une `ArgumentException` si besoin.

4.1.4 Affichage

Vous allez devoir afficher votre matrice !
Pour cela, vous allez utilisez la méthode `ToString` comme vu dans le cours (que vous avez lu bien sûr !).

```
1 public override String ToString();
```

Chaque ligne doit commencer et terminer par '|'. Il y aura un espace avant et après chaque élément de la matrice. Voici un exemple avec une matrice 5x5 initialisé avec la valeur 1 :

```
| 1 1 1 1 1 |
| 1 1 1 1 1 |
| 1 1 1 1 1 |
| 1 1 1 1 1 |
| 1 1 1 1 1 |
```

4.1.5 Accesseurs

Pour finir, nous voulons maintenant accéder aux valeurs de notre matrice sans accéder directement à ses attributs. Nous voulons donc faire directement `matrice[i, j]`. Grâce au C#, nous pouvons faire cela très simplement : en utilisant les propriétés. Pour cela vous avec donc à remplir la propriété suivante :

```
public T this[int i, int j]
{
    get
    {
        //FIXME
    }
    set
    {
        //FIXME
    }
}
```

Pour mieux comprendre comment ce type de propriété s'écrit, prenons comme exemple une classe générique `Array` qui n'aurait qu'un tableau à une seule dimension nommé `arr` comme attribut. Nous voulons pouvoir faire `objet[i]` afin d'accéder à l'élément du tableau. Dans ce cas la propriété s'écrira :

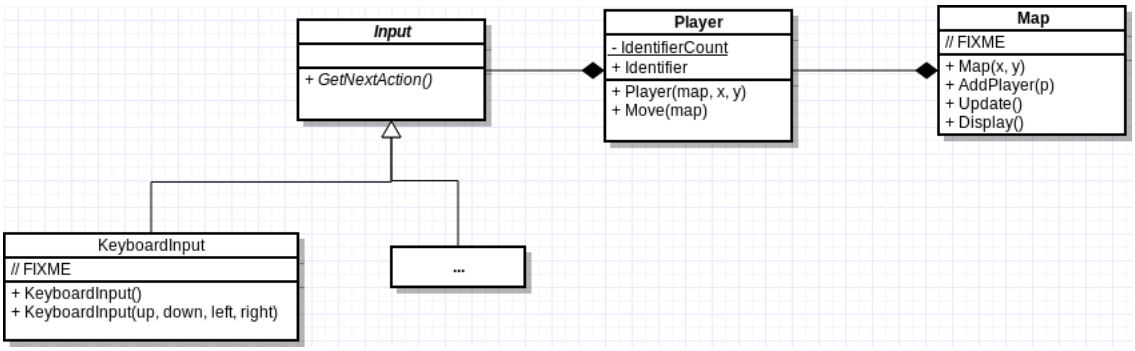
```
public T this[int i] //Vous pouvez ajouter un nombre
{                    //indéfini de variables
    get
    {
        return arr[i];
    }
    set
    {
        arr[i] = value;
    }
}
```

Maintenant à vous de jouez !

4.2 Tron

Bon, on pourrait continuer à faire plein de petits exercices sur la surcharge et les generics mais on est sympa avec vous. Le gros projet de cette semaine est tout autre chose ! Vous allez devoir coder un Light Cycle Game. Ce projet sera à coder dans le projet Tron.

Le projet devra suivre ce diagramme UML :



4.2.1 Quelques remarques

Pour cette partie du TP, nous allons moins vous guider que d'habitude. Vous devez implémenter chaque méthode demandée, mais vous aurez plus de facilité en ajoutant des attributs et méthodes auxiliaires. Vous devez utiliser la visibilité la plus restrictive : une méthode auxiliaire utilisée uniquement dans une autre méthode ne doit pas être publique.

Dans l'ensemble, le sujet sera souvent peu explicite sur la méthode à suivre. Du moment que le résultat à atteindre est clair, prenez le temps de réfléchir aux méthodes possibles avant de demander de l'aide à vos assistants.

4.2.2 Rappel des règles

La map commence vide, avec les joueurs répartis dessus. À chaque itération du jeu, tous les joueurs se déplacent dans une direction, la même qu'à l'itération précédente si aucune action n'est réalisée.

À chaque déplacement, les joueurs laissent un "mur" derrière eux. Si un joueur essaye de se déplacer sur un mur ou de sortir de la map, il est éliminé. Le gagnant est le dernier joueur présent sur la map.

4.2.3 Input

Démarrons par des choses simples, afin que notre jeu marche pour le mieux il faut gérer les entrées utilisateurs ! Nous voulons que notre programme fonctionne avec n'importe quelle entrée possible : clavier ou autre. Il nous faut donc représenter ces différentes entrées.

Pour cela nous allons utilisez une énumération représentant les différents types de mouvement (up, down, left and right). Elle va nous permettre d'associer le même type de mouvement pour n'importe quel type d'entrées utilisateurs.

Afin de représenter n'importe quel type d'entrée utilisateur, vous allez devoir coder une classe abstraite avec une seule méthode : `GetNextAction`. Elle renverra la prochaine action du joueur (up, down, right ou left). Le but de cette classe est de permettre la manipulation de tous les types d'input sans distinction.

```
public abstract class Input
{
        public enum Action {up, left, right, down};
        public abstract Action GetNextAction();
}
```

4.2.4 KeyboardInput

Pour ce TP, il faudra au minimum savoir gérer les entrées clavier. Il faudra donc créer une classe qui héritera de `Input`. Elle va pouvoir gérer n'importe quelle touche du clavier représentant les mouvements up, down, left et right. Vous devez permettre l'utilisation de deux constructeurs : le premier permet de donner les touches associées à chaque action, le second prend les flèches du clavier par défaut. Si le joueur n'a pas effectué d'action, la direction reste la même qu'au cycle précédent.

Faites bien attention à l'ordre des paramètres.

```
public KeyboardInput(ConsoleKey up, ConsoleKey down,
    ConsoleKey left, ConsoleKey right);
public KeyboardInput();
```

Cette classe est délicate à implémenter correctement, et pour éviter de vous bloquer au début du projet nous avons mis en place un système de paliers. Ils sont tous *obligatoires* jusqu'au 3 inclus, la suite en bonus. Vous pouvez vous arrêter sur un palier pour avancer sur le TP, et revenir dessus après avoir fini le reste. Vous ne pouvez pas utiliser le contenu de `System.Windows.Input`.
Les paliers sont les suivants :

1. Les entrées claviers sont lues. Ne vous arrêtez pas là.
2. Le jeu ne se bloque plus entre chaque cycle.
3. Plusieurs KeyboardInput peuvent être utilisés en parallèle pour plusieurs joueurs différents.
4. Le cas où chaque joueur appuie sur une touche pendant le même cycle doit être géré correctement.
5. KeyboardInput n'a plus besoin d'écrire du code à l'extérieur de la classe pour fonctionner, comme par exemple dans la boucle principale du jeu.

Hint : si vous êtes perdus, faites *l'inverse* des paliers bonus.

4.2.5 Player

Cette classe représente un joueur. Il doit avoir une position propre, et un identifiant permettant de le différencier des autres. Cet identifiant devra être un entier unique, assigné directement a l'objet au moment de son instanciation. Pensez aux attributs statiques.

Au moment de déplacer un joueur, il faut vérifier s'il s'est crash, ce qui arrive quand il sort de la map ou qu'il rentre dans un mur laissé par un joueur. Il faut également "marquer" la map pour indiquer qu'il y a maintenant un mur.

Chaque joueur a sa propre couleur. Elle s'applique à l'affichage du joueur en lui-même, ainsi qu'à ses murs. Vous pouvez gérer ces couleurs soit dans la classe Joueur, soit dans la classe Map. Vous devez avoir une couleur distincte pour au moins les 4 premiers joueurs, les joueurs suivants peuvent partager une couleur.

Vous devez implémenter au moins les méthodes suivantes :

```
1 public Player(Input input, int x, int y);
2 public bool Move(Map map);
```

4.2.6 Map

C'est la classe principale du projet. Elle doit gérer chaque joueur et l'état de la map en elle-même. `Update` doit déplacer chaque joueur d'une case dans la direction qu'il souhaite, vérifier s'il n'est pas mort, et vérifier la fin du jeu. La valeur de retour est l'identifiant du joueur gagnant, 0 si le jeu n'est pas fini. Pour l'affichage, vous devez afficher les bordures de la map en plus de son contenu.

Séparez bien l'affichage et la logique du jeu, ne touchez pas à `Console.Write` dans la méthode `Update()`, et inversement.

Vous devez implémenter au moins les méthodes suivantes, mais nous vous recommandons fortement d'en faire plus que ça. Nous ne voulons pas de méthode de plus de 50 lignes, factorisez votre code et séparez le en petites méthodes qui ont du sens.

```
1 public Map(int x, int y);
2 public void AddPlayer(Player p);
3 public int Update();
4 public void Display();
```

4.2.7 Main

Le main du jeu va initialiser la map et exécuter la boucle principale du jeu. Par défaut, le jeu se lance avec un joueur utilisant ZQSD et un autre les flèches pour se déplacer.

Lancer le jeu normalement sans argument doit directement lancer le jeu. Si vous voulez demander à l'utilisateur les dimensions du jeu, les contrôles des joueurs et autres, vous pouvez faire seulement si un argument est donné au programme (vous savez, les arguments de la fonction Main). L'idéal étant un fichier de configuration.

4.2.8 Impressionnez-nous !

Beaucoup de notions sont vu aujourd'hui. C'est pour cela que le TP est assez rapide. Nous attendons donc beaucoup de bonus de votre part ! Vous pouvez tout à fait rajouter des classes, rajouter d'autres inputs (souris, réseau ...). Si vous vous contentez pour l'affichage de clear et re-print, vous avez sans doute remarqué que c'est loin d'être fluide. Vous pouvez `SetCursorPosition` pour ré-afficher la map par dessus l'ancienne. Mieux, vous pouvez faire ça seulement sur les caractères qui ont changés. N'oubliez pas dans ce cas de remettre le curseur à la fin. Tout bonus sera à notifier dans le README sinon il ne sera pas noté.

The Code is the Law.