Contrôle 2 C# (Zoo)

1 Consignes de rendu

A la fin de ce contrôle, vous devrez soumettre une archive zip respectant l'architecture suivante :

```
prenom.nom.zip
    |-- prenom.nom/
        |-- AUTHORS
        |-- README
        |-- Preliminary/
            |-- Preliminary.sln
            |-- Preliminary/
                |-- tout sauf bin/ et obj/
        |-- Zoo/
            |-- Zoo.sln
            |-- Zoo/
                |-- 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 contrôle :

- Le fichier AUTHORS doit être au format habituel : *␣prenom.nom$ où le caractère '$' représente un retour à la ligne.
- Le fichier README n'est nécessaire que pour le dernier exercice.
- Vous devez suivre le sujet **scrupuleusement**, et notamment **respecter les prototypes** des différentes méthodes.
- Pas de dossiers bin ou obj dans le rendu final, ni aucun fichier binaire en général.
- **Le code doit compiler.**

2 Avant de commencer

Pour ce contrôle, vous devrez réaliser **les différentes parties dans le projet qui leur est associé** (c.f. l'architecture de rendu). Ces projets doivent être du type **Application Console**. Vous êtes, bien évidemment, libre de tester les méthodes de chaque partie dans votre **Main**. Avant de rendre, vous pouvez soit laisser vos tests, soit les enlever. Peu importe votre choix, nous remplacerons votre Main par le nôtre pour effectuer nos propres tests de correction. Enfin, vous avez tout à fait le droit d'implémenter des méthodes en plus de celles obligatoires.

3 Exercice 1 - Préliminaires

Cette première partie a pour but de vérifier si vous avez acquis les compétences de base du C#.

3.1 InitArray

Dans cet exercice, vous allez devoir initialiser un tableau tridimensionnel de taille n donnée en paramètre avec la valeur -1. Si n est négatif, vous devez lever une exception *ArgumentOutOfRangeException* avec un message approprié.

```
1 static short[,,] InitArray(int n);
```

3.2 IntToBin

Dans cet exercice, vous allez devoir convertir l'entier donné en paramètre en binaire.
Là encore, toutes les fonctions de conversion sont interdites. Seul les nombres positifs doivent êtres gérés. Dans le cas d'un nombre négatif, une exception *ArgumentException* doit être déclenchée avec un message approprié.

```
1 static string IntToBin(int n);
```

Exemples

```
1 IntToBin(42); // "101010"
2 IntToBin(256); // "100000000"
```

3.3 BinToInt

Dans cet exercice, vous allez devoir convertir la chaîne de caractère donnée en paramètre en un entier. La chaîne de caractère correspond à un nombre en binaire. Le premier caractère de la chaîne de caractère (celui en position 0) représente le bit de poids fort du nombre. Dans le cas de la chaîne de caractère vide, renvoyer 0.
Aucune fonction prédéfinie (fonctions de conversion et autres) n'est autorisée. Les nombres négatifs ne seront pas testés.
Les cas d'erreurs doivent déclencher une exception *ArgumentException* avec un message approprié.

```
1 static int BinToInt(string str);
```

Exemples

```
1 BinToInt("101010"); // 42
2 BinToInt("100000000"); // 256
```

3.4 MyAtoi

Dans cet exercice, vous allez devoir transformer une chaine de caractères donnée en paramètre en entier.
Vous n'avez évidemment pas le droit d'utiliser les fonctions de conversion (*int.Parse()* par exemple)
Les nombres négatifs doivent également être convertis. Dans le cas de la chaine de caractère vide, renvoyer 0. Les cas d'erreurs (des lettres dans la chaine de caractères par exemple) doivent déclencher une exception *ArgumentException* avec un message approprié.

```
static int MyAtoi(string s);
```

Exemples

```
MyAtoi("42"); // 42
MyAtoi("35383773"); // 35383773
```

3.5 MatrixMult

Dans cet exercice, vous allez devoir implémenter le produit matriciel des 2 matrices données en paramètre.

Le produit matriciel est défini comme suivant :
Si $A = (a_{ij})$ est une matrice de type (m, n) et $B = (b_{ij})$ est une matrice de type (n, p), alors le produit, noté $AB = (c_{ij})$ est une matrice de type (m, p) donnée par :

$$\forall i, j : c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Si le produit matriciel entre les 2 matrices données en paramètre n'est pas possible, une exception *ArgumentException* doit être déclenchée avec un message approprié.

```
static int[,] MatrixMult(int[,] A, int[,] B);
```

Exemples

```
int[,] A = {
            { 1, 2, 3 },
            { 4, 5, 6 },
            { 7, 8, 9 }
           };

int[,] B = {
            { 9, 8, 7 },
            { 6, 5, 4 },
            { 3, 2, 1 }
```

```
            };

int[,] C = MatrixMult(A, B);
/*
    Value of C : 30  24  18
                 84  69  54
                 138 114 90
*/

int[,] D = MatrixMult(B, A);
/*
    Value of D : 90 114 138
                 54 69  84
                 18 24  30
*/
```

4 Exercice 2 - Zoo

Cet exercice porte sur la structure d'un programme. Il est conseillé de le lire en entier avant de commencer car cela permet de mieux comprendre l'exercice et donc de prendre de meilleures décisions d'implémentation.

Cet exercice a pour but de construire une hiérarchie de classes simples permettant de gérer des animaux et leur enclos. L'implémentation des classes demandées est libre (attributs, méthodes et classes abstraites peuvent êtres ajoutés), mais vous devrez respecter les prototypes demandés et toujours utiliser l'accessibilité la plus restrictive (*public*, *private*, *protected*) ainsi que des accesseurs lorsqu'ils sont nécessaires.

4.1 Caractéristiques des animaux

On distinguera les animaux au sein d'une même espèce par certaines caractéristiques représentées par la classe abstraite *BodyFeature*. Toutes les classes représentant une ou plusieurs caractéristiques hériteront de *BodyFeature*.

Vous devez implémenter les classes suivantes (héritant directement ou indirectement de *BodyFeature*) :

— *EyeColor* : représente la couleur des yeux par l'une des 15 couleurs de *ConsoleColor*

— *HairColor* : représente la couleur du poil par l'une des 15 couleurs de *ConsoleColor*

— *Height* : représente la taille de l'animal par un entier (en centimètres)

Vous pouvez ajouter des classes abstraites intermédiaires si vous le souhaitez. Vous devez faire un constructeur pour chacune de ces 3 classes. Il faut lever une exception *ArgumentOutOfRangeException* si une taille strictement inférieure à 2 est donnée au constructeur de *Height*. Vous devez redéfinir la méthode *ToString()* (*public override string ToString()*), la méthode doit retourner la chaîne de caractères suivante :

— *EyeColor* : "Eye Color : [color]" où [color] est la ConsoleColor
— *HairColor* : "Hair Color : [color]" où [color] est la ConsoleColor
— *Height* : "Height : [h]cm" où [h] est la valeur de la taille en centimètres

Attention, la méthode *ToString()* ne doit rien afficher sur aucune sortie, elle ne fait que retourner une chaîne de caractère. Notez aussi qu'il n'y a pas de retour à la ligne à la fin de la chaîne de caractères.

4.2 Animaux

Les animaux sont représentés par des classes héritant de la classe abstraite *Animal*. Tous les animaux doivent avoir un nom (chaîne de caractères) ainsi que 3 caractéristiques (1 pour chaque classe implémentée précédemment). La classe abstraite *Predator* doit hériter de *Animal*.

Pour cet exercice, on implémente 2 espèces :

— Phacochère : classe *Warthog* héritant de *Animal*
— Lion : classe *Lion* héritant de *Predator*

Ces classes doivent avoir un constructeur (arguments : *name, eyecolor, haircolor, height*) et doivent aussi redéfinir la méthode *ToString()*. Le format de la chaîne de caractère est le suivant ($ représente un retour a la ligne).

```
------------------------$
| [species]$
| Name : [name]$
| Eye Color : [color]$
| Hair Color : [color]$
| Height : [h]cm$
| Predator : [Yes/No]$
------------------------
```

Notez qu'il n'y a pas de retour à la ligne à la fin de la chaîne. La première ligne et la dernière ligne ont une longueur de 24 caractères. Bien entendu, vous devez remplacer [species] par l'espèce, [name] par le nom de l'animal, [color] par une couleur, [h] par une taille et [Yes/No] par Yes ou No.

On aura besoin de la méthode *Breed(Warthog other, string childName)* dans la classe *Warthog* pour l'exercice suivant. Cette méthode doit retourner un nouveau phacochère avec le nom donné en argument. Chacune des caractéristiques de l'enfant est choisie aléatoirement entre celles de ses 2 parents (*this* et *other*).

4.3 Enclos à phacochères

On veut simuler un enclos à phacochères. La structure du code (classe et prototypes de méthodes à respecter) vous est donnée à la fin du sujet. Il est fortement conseillé de lire l'exercice en entier avant de commencer à coder.

L'enclos est tableau de phacochères à 2 dimensions.

Conseil : Un emplacement vide de l'enclos peut être représenté par *null*.

Le constructeur doit lever une exception *ArgumentOutOfRangeException* avec un message approprié si l'une des dimensions est négative ou nulle. Il ne doit pas mettre d'animal dans l'enclos.

La méthode *AddWarthog* doit placer le phacochère aux coordonnées demandées dans l'enclos et retourner vrai. Si les coordonnées ne correspondent pas à un emplacement de l'enclos, elle doit lever une exception *ArgumentOutOfRangeException*. Si l'emplacement demandé est déjà occupé, elle ne doit que retourner faux (ne pas modifier l'enclos).

La méthode *Display* doit afficher l'enclos sur la sortie standard comme suit (13x4) :

```
----------P-
--W---WW-----
-WWW--WWWWW--
--WWWWW------
```

Chaque ligne se termine par un retour à la ligne y compris la dernière. Un emplacement vide est représenté par le caractère '-' **blanc**. Un phacochère est représenté par le caractère 'W' **de la couleur de sa caractéristique *HairColor***. Le prédateur (s'il y en a un) est représenté par le caractère 'P' **de la couleur de sa caractéristique *HairColor***

On veut simuler l'évolution de la population dans l'enclos. La méthode *Step* sera appelée à chaque fois que l'on veut mettre à jour l'état de l'enclos. Elle est décomposée en 3 méthodes privées dont 2 qui ne sont appelées que lorsqu'un prédateur est introduit dans l'enclos.

La méthode *StepWarthogs* définit l'évolution des phacochères en une étape de la simulation. Le voisinage d'un phacochère est l'ensemble des 4 emplacements ou moins étant directement au nord, à l'ouest, au sud et à l'est (donc sans les diagonales). Chaque phacochère (déjà présent dans l'enclos avant l'appel à *StepWarthogs*) qui a au moins 1 voisin, a 20% de chances de se reproduire. S'il se reproduit, chacun des emplacements vides de son voisinage est rempli par le résultat de l'appel à la méthode *Breed* (un appel par emplacement vide) avec pour arguments :

— *other* : L'un de ses voisins choisi aléatoirement parmi ceux déjà présents dans l'enclos avant l'appel à *StepWarthogs*.

— *name* : La chaîne de caractères "x-y" où x et y sont les coordonnées du nouveau phacochère.

Conseil : Pour que les phacochères qui n'étaient pas présents avant l'appel à *StepWarthogs* n'interfèrent pas, on peut utiliser un tableau temporaire afin de préserver l'état actuel de l'enclos et remplacer l'ancien tableau par le nouveau à la fin.

La méthode *StepPredator* déplace le prédateur. Les mouvements en diagonale sont autorisées. Le but du prédateur est de manger les autres animaux. Le prédateur ne doit jamais se déplacer de plus d'un emplacement à la fois et ne doit pas sortir de l'enclos. Idéalement, le prédateur cherche à manger le plus d'animaux possibles. Toutefois, une solution peu optimisée qui ne fait pas de mouvements invalides suffira. Vous devez expliquez la stratégie de votre prédateur en une phrase dans le README.

Conseils : Une stratégie possible est de faire des allers et retours dans l'enclos, en passant par tous les emplacements ou non. Une autre stratégie facile à implémenter est de faire se déplacer le prédateur aléatoirement dans l'enclos. Ne cherchez pas quelque chose de trop compliqué.

La méthode *PredatorFeed* enlève de l'enclos (assignation à *null*) les animaux à portée du prédateur. Le prédateur peut atteindre au plus 9 emplacements : celui où il est ainsi que les 8 emplacements adjacents (ou moins). Les diagonales sont donc incluses.

Vous trouverez sur la page suivante la structure de la classe ainsi que les prototypes des méthodes à respecter. Vous pouvez ajouter des attributs et méthodes privés si nécessaire.

```
class WarthogEnclosure
    {
        // You may add your attributes here ...
        public Predator preda { private get; set; } = null;
        private int predaX = 0;
        private int predaY = 0;

        public WarthogEnclosure(int width, int height) { /* ... */ }
        public bool AddWarthog(Warthog w, int x, int y) { /* ... */ }

        public void Display()
        {
            ConsoleColor oldColor = Console.ForegroundColor;
            /* Your code ... */
            Console.ForegroundColor = oldColor;
        }

        private void StepWarthogs() { /* Your code ... */ }
        private void StepPredator() { /* Your code ... */ }
        private void PredatorFeed() { /* Your code ... */ }

        public void Step()
        {
            StepWarthogs();
            if (preda != null)
            {
                StepPredator();
                PredatorFeed();
            }
        }
    }
```

The code is the law.