TP C#6 : WestWorld Tycoon

Consignes de rendu

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

```
rendu-tp-prenom.nom.zip
+-- prenom.nom/
    |-- AUTHORS
    |-- bot.out (auto-generated by your program)
    |-- README
    +-- WestWorldTycoon/
        |-- WestWorldTycoon/
        |   +-- Everything except bin/ and obj/
        +-- WestWorldTycoon.sln
```

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).

Table des matières

1 Introduction **3**
1.1 Objectifs . . . 3

2 Cours **3**
2.1 Rappel . . . 3
2.1.1 Classes et Objets . . . 3
2.1.2 Encapsulation . . . 3
2.1.3 Constructeurs et Destructeurs . . . 4
2.2 Visibilité . . . 4
2.2.1 Static . . . 4
2.2.2 public vs private . . . 5
2.2.3 protected . . . 6
2.3 Héritage . . . 7
2.3.1 abstract et sealed . . . 9
2.4 Polymorphisme . . . 10
2.4.1 Surcharge . . . 10
2.4.2 Substitution . . . 11

3 Exercice : WestWorld Tycoon **14**
3.1 Introduction . . . 14
3.1.1 Objectifs . . . 14
3.1.2 Description du jeu . . . 14
3.2 Basics . . . 14
3.2.1 Attraction . . . 14
3.2.2 Shop & House . . . 16
3.2.3 Tile . . . 17
3.2.4 Map . . . 18
3.2.5 Game . . . 19
3.3 Bot . . . 21
3.3.1 Start . . . 21
3.3.2 Update . . . 22
3.3.3 End . . . 22
3.3.4 Votre propre bot . . . 22
3.4 Bonus . . . 24
3.4.1 Destroy . . . 24
3.4.2 Map copy . . . 24

1 Introduction

1.1 Objectifs

Dans ce TP, nous allons approfondir les notions abordées durant le TP C#4. Il vous sera demandé de coder et comprendre les interactions entre objets au sein d'un jeu video assez simple. Les notions importantes de ce sujet sont les suivantes :

— Classe, Objet
— Constructeur et Destructeur
— Static
— Public, Private
— Héritage
— Surcharge et Override

2 Cours

Les notions liées à la programmation orientée objet peuvent être extrêmement complexes, surtout l'interaction entre toutes ces notions. Il est impératif que vous en maîtrisiez certaines, mais d'autres ne sont présentées que pour les plus curieux.

2.1 Rappel

Nous presentons ici les notions abordées pendant le TP C#4. Ces notions doivent être comprises.

2.1.1 Classes et Objets

C# est un langage orienté objet. Cela signifie qu'il est possible de définir et d'instancier ses propres **objets**. Mais qu'est-ce qu'un **objet** exactement ?

Un **objet** est une structure de données décrite par la **classe** à laquelle l'objet appartient. Attention, ces deux notions ne sont pas équivalentes. Les termes **classe** et **objet** ne peuvent pas être interchangés. La **classe** décrit la structure et le comportement de l'**objet**. En C#, la **classe** sert de type comme `int` ou `float`. La variable sera alors considérée comme étant un **objet**.

```
1  class MyClass //ceci est une classe
2  {
3  }
4
5  public static void Main(string[] args)
6  {
7    MyClass foo = new MyClass(); //ceci est un objet
8  }
```

Dans l'example précédent, `foo` est un **objet** de **classe** `MyClass`. Mais il existe d'autres façons de décrire la relation qui lie MyClass à `foo` :

— `foo` est une **instance** de `MyClass`.
— `MyClass` est le type de la variable `foo`.

2.1.2 Encapsulation

Les **classes** servent à regrouper et lier des informations (**Attributs**) et comportements (**Méthodes**). Cela s'appelle `l'encapsulation`.

```
1  class MyClass
2  {
3    int myInt; //ceci est un attribut
4    string myString;
5    float myFloat;
6
7    void PrintMyString() //ceci est une méthode
8    {
9      Console.WriteLine(myString);
10   }
11 }
```

Les **attributs** décrivent les informations contenues dans l'**objet** et les **méthodes** décrivent la manière d'interagir avec ces informations. Bien évidemment, il est possible de modifier les **attributs** d'un **objet** depuis une **méthode**.

2.1.3 Constructeurs et Destructeurs

Les **constructeurs** et **destructeurs** servent à définir la durée de vie d'un **objet**. Un **objet** naît à l'appel de son **constructeur** et meurt à l'appel de son **destructeur**. Un **constructeur** n'a pas de valeur de retour et porte toujours le nom de la **classe**.

```
1  class MyClass
2  {
3    int myInt;
4
5    public MyClass(int i) //ceci est un constructeur
6    {
7      this.myInt = i;
8    }
9  }
```

Le **constructeur** sert entre autre à donner aux **attributs** d'un **objet** leur valeur par défaut. Le mot clef `this` réfère a l'**objet** lui même et permet de distinguer les **attributs** des paramètres d'une **méthode**.

Vous n'aurez pas à déclarer de **destructeur**, car C# en fournit un par défaut (un **constructeur** par defaut est aussi fourni par C#). Il sert à nettoyer l'environnement (mémoire, etc) à la destruction de l'**objet**.

2.2 Visibilité

Vous vous demandez sûrement ce que signifient les mots clefs : `static`, `public` et `private`. C'est aujourd'hui que nous allons vous l'expliquer. Ces trois notions doivent être comprises.

2.2.1 Static

Commencons par le mots clef `static`. Il signifie que l'**attribut** ou **méthode** qui suit ne dépend pas de l'**objet** en lui-même mais uniquement de la **classe** à laquelle l'**objet** appartient.

```
1  class Human
2  {
3    static int globalPopulation = 0;
4    Human()
5    {
6      ++globalPopulation;
7    }
8  }
9  public static void Main(string[] args)
10 {
11   var h1 = new Human();
12   var h2 = new Human();
13
14   Console.WriteLine(Human.globalPopulation) // globalPopulation = 2;
15   //h1.globalPopulation      instruction illégale
16 }
```

Ainsi, l'**attribut** globalPopulation n'appartient plus à un **objet** mais à la **classe** Human. C'est pourquoi il n'est plus possible d'y accéder à travers un **objet** ou le mot clef `this`. Il faut passer par le nom de la **classe**.

Une **méthode** `static` ne peut accéder et modifier que les **attributs** `static` de la **classe**. De plus, si une **classe** est marquée comme `static`, tous ses **attributs** et toutes ses **méthodes** doivent l'être également. C'est le cas de la **classe** `Console` par exemple.

2.2.2 public vs private

Les mots clefs `public` et `private` servent à définir les contraintes d'accès aux **attributs** et aux **méthodes** de l'**objet**. `public` indique qu'il est possible d'accéder à l'**attribut/méthode** avec l'opérateur `.` directement depuis l'**objet** (ex : `obj.attribut`). `private` indique que seul l'**objet** lui-même peut accéder à son **attribut/méthode** depuis ses autres **méthodes**. Si aucun mot clef n'est spécifié, c'est `private` qui est utilisé par défaut.

```
1  class MyClass
2  {
3    public string myPublicString;
4    private string myPrivateString;
5
6    public string myReadOnly
7    {
8      // Getter
9      get { return myPrivateString;}
10   }
11   public string myWriteOnly
12   {
13     // Setter
14     set { myPrivateString = value; }
15   }
16   public string myReadButNotWrite
17   {
18     get { return myPrivateString;}
19     private set { myPrivateString = value; }
20   }
21 }
```

Le mot clef `private` est important pour garantir une certaine sécurité dans un programme. Il permet entre autres de vous empêcher de modifier la taille d'une chaîne de caractères à la main.

Il est également possible de définir des attributs **read only** ou **write only** grâce à un système de **getter/setter**.

2.2.3 protected

Il existe un autre mode de protection : `protected`. Ce mot clef protège un **attribut/méthode** de la même façon que `private` à une exception près. L'**attribut/méthode** est considéré comme `public` dans toutes les **classes** héritant de l'**attribut/méthode** (l'héritage est expliqué dans la partie suivante).

```
1  class MyClass
2  {
3    private string myPrivateString;
4    protected string myProtectedString;
5
6    public MyClass(string myPrivateString)
7    {
8      this.myPrivateString = myPrivateString;
9      this.myProtectedString = myPrivateString;
10   }
11 }
12
13 class MySubClass : MyClass
14 {
15   public MySubClass(string myPrivateString, string myProtectedString)
16   : base(myPrivateString)
17   {
18       //this.myPrivateString = myPrivateString;  instruction illégale
19       this.myProtectedString = myProtectedString;
20   }
21 }
```

Encore une fois, ce mot clef est utile pour sécuriser une application.

2.3 Héritage

L'héritage est l'un des points les plus importants de la programmation objet. Cela permet de ne pas dupliquer de code mais surtout de partager des origines communes entre **classes**. La **classe** qui hérite est appelée **classe fille** ou **sous-classe** et celle dont la elle hérite est appelée **classe mère** ou **super-classe**.

```
Item
|-- Equipement
|    |-- Armure
|    |-- Arme
|    |-- ...
|-- Consommable
     |-- Nouriture
     |-- ...
```

Mais l'héritage ne se limite pas à une simple généalogie. Un second point important est que toute **classe fille** peut être utilisée à la place de sa **classe mère** (ceci sera très très utile pour votre projet de S2).

Un petit exemple :

```
1  class Student
2  {
3    protected int promotion;
4    protected string name;
5
6    public Student(int promotion, string name)
7    {
8      this.promotion = promotion;
9      this.name = name;
10   }
11
12   public int GetPromo()
13   {
14     return this.promotion;
15   }
16 }
17
18 class Sup : Student
19 {
20   public bool sharp;
21
22   public Sup(int promotion, string fullname, bool isSharp)
23   {
24     this.promotion = promotion;
25     name = fullname;
26     this.sharp = isSharp;
27   }
28 }
29
30 class Ing1 : Student
31 {
32   public bool assistant;
33
34   public Ing1(int promotion, string name, bool ACDC, bool ASM)
35   : base (promotion, name)
36   {
37     assistant = ACDC || ASM;
38   }
39 }
```

Comme vous pouvez le voir, toute **sous-classe** peut accéder aux **attributs** et **méthodes** (`public` et `protected`) de sa **super-classe** directement (avec ou sans `this`). De plus `base` réfère à la **méthode** équivalente de la **super-classe** (nous en reparlerons un peu plus tard).

```
1  public static void Main(string[] args)
2  {
3    Sup b2 = new Sup("John Doo", 2022, false);
4    Ing1 c1 = new Ing1("Julien Mounier", 2020, true, false);
5    Student a1 = new Ing1("Florian Amsallem", 2020, true, false);
6    b2.GetPromo();
7    c1.GetPromo();
8    a1.GetPromo();
9    //b2.assistant; instruction illégale
10   //a1.assistant; instruction illégale
11 }
```

Attention, il est possible de faire une convertion implicite vers une **super classe** mais pas l'inverse. C'est pourquoi la ligne 5 est valide contrairement à la 10. Même si a1 à été instancié en tant qu'Ing1, sa déclaration en tant que Student ne lui donne pas accès aux **attributs** propres à la **classe** Ing1. Il est par contre possible de tester si un objet appartient bien à une **classe** grâce au mot clef `is`.

```
1  public static void Main(string[] args)
2  {
3    Student a1 = new Ing1("Florian Amsallem", 2020, true, false);
4    if (a1 is Ing1) // ici vrai
5      // do something
6  }
```

2.3.1 abstract et sealed

Il est possible qu'une **classe** n'ait pas pour objectif d'être **instanciée**. Elle peut servir à définir une structure commune pour que d'autres **classes** puissent en hériter. Le mot clef `abstract` empêche que des **objets** soient créés à partir d'une telle **classe**.

Il est également possible d'empêcher toute **classe** d'hériter d'une **classe** A. Il faut alors indiquer que la **classe** A est `sealed`.

```
1  abstract class MCQ<T>
2  {
3    protected string question;
4    protected int answer;
5    protected T[] choices;
6    public bool IsCorrect(int choice)
7    {
8      return choice == answer;
9    }
10
11   public void AskQuestion()
12   {
13     Console.WriteLine(question);
14     foreach (T choice in choices)
15       Console.WriteLine(choice);
16   }
17 }
18
19 sealed class MathMCQ : MCQ<int> { }
20 sealed class FrenchMCQ : MCQ<string> { }
```

Ce petit exemple démontre à la fois l'utilité des mots clefs `abstract` et `sealed`, et sert de premier contact avec la **généricité** (`Class<Type>`). Il n'est vraiment pas nécéssaire que vous maîtrisiez la **généricité** à la fin de ce TP.

Les mots clefs `abstract` et `sealed` peuvent également être appliqués à un unique **attribut** ou **méthode**.

2.4 Polymorphisme

Le **polymorphisme** est le dernier point important en progammation orientée objet. Cela signifie qu'un **objet**, **méthode** ou **attribut** peut changer de forme au cours de l'exécution. Nous vous en avons déjà parlé sans que vous vous en rendiez compte. Le fait qu'un **objet** puisse prendre le type de sa **super-classe**, c'est du **polymorphisme**. Cela peut vous sembler assez faible puisque presque toutes les **méthodes** et **attributs** d'une **super-classe** sont accessibles dans une **sous-classe**.

Remonter dans l'arbre généalogique d'une **classe** reviendrait donc à perdre l'accès aux **attributs** et **méthodes** introduits par les **sous-classe**. C'est vrai, mais ce n'est pas tout, car il est possible de redéfinir une **méthode** au moment de l'héritage. Dans ce cas, le polymorphisme permet donc d'avoir accès à la version initiale de la **méthode**.

Il est important que vous compreniez les prochaines notions.

2.4.1 Surcharge

Commençons simplement par la **surcharge**. C'est ce qui vous permet de donner un nom identique à deux fonctions/méthodes. Il y a quand même quelques conditions à respecter :

— Toutes les fonctions doivent avoir une **signature** différente. C'est-à-dire que l'association type de retour, nom de fonction et type des paramètres (dans l'ordre où ils arrivent) doit être unique ;

— Un changement du type de retour uniquement n'est pas suffisant (au moins un des paramètres doit changer).

Plus besoin de suffixer ou préfixer vos fonctions récursives par exemple. La surcharge est également beaucoup utilisée afin de créer plusieurs **constructeurs** pour une **classe**.

```
1  public int sum(int[] arr, int pos)
2  {
3    if (pos < arr.Length)
4      return arr[pos] + sum(arr, pos + 1);
5    return 0;
6  }
7
8  public int sum(int[] arr)
9  {
10   return sum(arr, 0);
11 }
```

```
1  public class vector2
2  {
3    public int x;
4    public int y;
5  }
6
7  public class Point
8  {
9    private vector2 pos;
10
11   public Point(Point p)
12   {
13     Point(p.pos);
14   }
15
16   public Point(vector2 vec)
17   {
18     Point(vec.x, vec.y);
19   }
20
21   public Point(int x, int y)
22   {
23     this.pos.x = x;
24     this.pos.y = y;
25   }
26 }
```

2.4.2 Substitution

La **substitution** consiste à changer la définition d'une **méthode** ou d'un **attribut**. Et pour cela, deux possibilités :

— on écrase la version précedente avec `new` ;
— on l'adapte avec `override`.

Mais seules les **méthodes** marquées avec `virtual`, `override` ou `abstract` peuvent être redéfinies grâce à `override` dans une **classe fille** alors que `new` peut tout écraser.

Attention, toute **méthode** marquée avec `abstract` doit être `override` dans toutes les **classes filles**.

```
1  class Student
2  {
3    protected int promotion;
4    protected string name;
5
6    public Student(int promotion, string name)
7    {
8      this.promotion = promotion;
9      this.name = name;
10   }
11
12   public int GetPromo()
13   {
14     return this.promotion;
15   }
16
17   public virtual void SayHi()
18   {
19     Console.WriteLine("I'm " + name);
20   }
21 }
```

```
1  class Sup : Student
2  {
3    public bool sharp;
4
5    public Sup(int promotion, string fullname, bool isSharp)
6    {
7      this.promotion = promotion;
8      name = fullname;
9      this.sharp = isSharp;
10   }
11
12   public override void SayHi()
13   {
14     base();  // Console.WriteLine("I'm " + name);
15     if (sharp)
16       Console.WriteLine("And I'm in sharp");
17   }
18 }
```

Avec `override`, il est possible d'accéder à la **méthode** de la **super-classe** grâce au mot clef `base`. Ce mot clef est souvent utilisé dans les **constructeurs**.

```
class Ing1 : Student
{
  public bool assistant;

  public Ing1(int promotion, string name, bool ACDC, bool ASM)
  : base (promotion, name)
  {
    assistant = ACDC || ASM;
  }

  public override void SayHi()
  {
    Console.WriteLine("Hello.\n);
    Console.WriteLine("My name is " + name + ".\n");
    Console.WriteLine("Have a good day.");
  }

  public new int GetPromo()
  {
    return 2020;
  }
}
```

Ce sont ces mécanismes qui permettent d'avoir plusieurs fonctions avec exactement la même **signature** mais des comportements différents. Attention cependant, il n'est pas possible d'accéder à toutes les définitions d'une même **méthode** a partir d'un seul **objet** directement. La définition choisie par le compilateur dépend du type actuel de l'**objet**. Il est donc possible grâce à des convertions de type de choisir la définition qui nous intéresse.

```
public static void Main(string[] args)
{
  Ing1 c1 = new Ing1("Julien Mounier", 2020, true, false);
  c1.SayHi();
  /*
  Hello.

  My name is Julien Mounier.

  Have a good day.
  */
  ((Student) c1).SayHi(); // convertit c1 en Student puis appel SayHi.
  /*
  I'm Julien Mounier
  */
}
```

Attention, une **méthode** `override` ne peut pas perdre en protection, elle ne peut que gagner. Il est possible d'`override` de `public` vers `private` mais pas l'inverse. `protected` se trouve entre les deux.

3 Exercice : WestWorld Tycoon

3.1 Introduction

Félicitations ! Vous êtes candidat pour devenir manager de **WestWorld** ! Pour sélectionner le meilleur manager d'entre vous, nous avons préparé un test. Vous allez devoir gérer un parc d'attraction virtuel : *WestWorld Tycoon*.

WestWorld Tycoon est un jeu de gestion simulant un parc d'attraction. Dans ce jeu, vous pouvez construire des bâtiments comme des magasins, des maisons et bien sûr des attractions. Votre but est de récolter un maximum d'argent dans un temps fini. Le candidat ayant récolté le plus d'argent sera recruté comme manager du vrai parc.

3.1.1 Objectifs

Dans cet exercice, vous allez devoir implémenter le jeu *WestWorld Tycoon* puis un programme pour gérer le parc le mieux possible (un bot). Pour cela, vous disposez d'une structure disponible sur l'intranet. De manière à être noté, vous devez impérativement suivre la structure.

De manière générale, nous souhaitons que le bot ne puisse pas tricher. Par exemple, il ne doit pas pouvoir modifier l'argent restant.

3.1.2 Description du jeu

Le parc est représenté par une matrice (la carte). La carte possède trois types de biomes différents :
- Mer "*Sea*" : Une étendue d'eau qui bloque la construction de bâtiments.
- Montagne "*Mountain*" : Des montagnes qui bloquent la construction de bâtiments.
- Plaine "*Plain*" : Une zone qui autorise la construction de bâtiments.

Le bot peut réaliser trois actions :
- Construire un bâtiment "*Build*" : Cela peut se faire uniquement sur un terrain libre de type plaine.
- Améliorer un bâtiment "*Upgrade*" : Chaque bâtiment peut être améliorer pour améliorer son effet.
- Détruire un bâtiment "*Destroy*" : Chaque bâtiment peut être détruit gratuitement.

Il existe trois bâtiments différents :
- Attraction "*Attraction*" : Permet d'augmenter la population du parc.
- Maison "*House*" : Permet d'augmenter la capacité maximale du parc.
- Magasin "*Shop*" : Permet de gagner de l'argent en fonction de la population du parc.

3.2 Basics

Cette partie est nécessaire pour passer à la suivante. Le but est d'implémenter les règles du jeu afin de coder un Bot qui devra avoir le score le plus grand possible.

3.2.1 Attraction

Vous pouvez ouvrir le fichier `Attraction.cs` ! Vous pouvez voir deux choses :
- La classe `Attraction` hérite de la classe `Building` qui est déjà implémentée.

— La classe `Attraction` possède des constantes comme le prix de construction et d'amélioration. **Ne modifiez pas ces valeurs**.

Vous allez commencer par ajouter un **attribut** `lvl` à la classe. Celui-ci est de type `int` et doit avoir une protection **private** pour empêcher la modification du niveau d'une attraction depuis l'extérieur de la classe.

Vous pouvez maintenant implémenter le constructeur de la classe. Celui-ci doit initialiser tous ses attributs (y compris ceux de sa classe mère).

```
1  public Attraction()
2  {
3      //TODO
4  }
```

Vous devez ajouter un **getter** pour pouvoir accéder au niveau d'une attraction (il ne faut pas pouvoir modifier celui-ci).

```
1  public int Lvl
2  {
3      //TODO
4  }
```

Il ne vous reste plus qu'a implémenter les méthodes `Upgrade` et `Attractiveness`.

La méthode `Upgrade` permet d'améliorer le niveau de l'attraction. Elle prend en paramètre une référence vers un entier `money` qui correspond à l'argent restant du parc. La méthode doit renvoyer `true` si l'amélioration est possible. De plus, si celle-ci est possible, elle doit retrancher le coût de l'amélioration à `money`.

```
1  public bool Upgrade(ref int money)
2  {
3      // TODO
4  }
```

Conseil : Vous devez utiliser le tableau constant `UPGRADE_COST` déclaré plus haut. Celui-ci correspond aux prix de chaque amélioration. Il ne comporte que trois valeurs car une attraction ne peut pas être améliorée plus de trois fois.

Pour finir, implémentez la méthode `Attractiveness` qui renvoie un `long` qui correspond à l'attractivité que génère le bâtiment. Cette valeur dépend du niveau de l'attraction.

```
1  public long Attractiveness()
2  {
3      // TODO
4  }
```

Conseil : Vous devez utiliser le tableau constant `ATTRACTIVENESS`. Par exemple, au niveau zéro, une attraction va générer une attractivité de `ATTRACTIVENESS[0]`.

3.2.2 Shop & House

Vous pouvez faire exactement la même chose pour les classes `Shop` et `House`.

```
1  public Shop()
2  {
3      // TODO
4  }
5
6  public int Lvl
7  {
8      // TODO
9  }
10
11 public long Income(long population)
12 {
13     // TODO
14 }
15
16 public bool Upgrade(ref int money)
17 {
18     // TODO
19 }
```

Le revenu d'un magasin donné par `Income` dépend du niveau du magasin. Le tableau constant `INCOME` vous donne le pourcentage de la population à faire un achat dans le magasin. On considère qu'un achat fait gagner 1 dollar.

```
1  public House()
2  {
3      // TODO
4  }
5
6  public int Lvl
7  {
8      // TODO
9  }
10
11
12 public long Housing()
13 {
14     // TODO
15 }
16
17 public bool Upgrade(ref int money)
18 {
19     // TODO
20 }
```

La "capacité en visiteurs" donné par `Housing` dépend du niveau de la maison. Cette valeur est directement donnée par le tableau constant `HOUSING`. Par exemple, une maison de niveau zéro génère une capacité de `HOUSING[0]` visiteurs.

3.2.3 Tile

Nous allons maintenant nous occuper des tuiles de notre carte "*tile*".

Une tuile possède deux attributs :
— Un biome "*biome*"
— Un bâtiment "*building*" dont la valeur est `null` si aucun bâtiment n'est construit.

Implémentez les deux attributs de la classe `Tile`. Ces attributs ont une protection `private`. Ils ne doivent pas être **modifiables** depuis l'extérieur de la classe.

Vous pouvez implémenter le constructeur qui prend un `Biome` en paramètre. Le bâtiment doit être initialisé à `null`.

```
1  public Tile(Biome b)
2  {
3      // TODO
4  }
```

Vous devez ajouter un **getter** pour pouvoir accéder au biome d'une tuile (il ne faut pas pouvoir modifier celui-ci).

```
1  public Biome GetBiome
2  {
3      // TODO
4  }
```

On peut maintenant ajouter des méthodes à notre classe. La première méthode est `Build`, elle prend en paramètre une référence vers un entier `money` qui correspond à l'argent restant du parc et `type` le type de bâtiment que l'on souhaite construire. La méthode doit retourner `true` si la construction est possible et doit créer un nouveau bâtiment du type souhaité et retrancher le prix de construction à l'argent disponible. Rappelez-vous que l'on peut construire des bâtiments uniquement sur un biome de type plaine.

```
1  public bool Build(ref int money, Building.BuildingType type)
2  {
3      // TODO
4  }
```

La deuxième méthode que l'on va implémenter est `Upgrade`. Elle prend en paramètre une référence vers un entier `money` qui comme précédemment correspond à l'argent restant du parc. Si l'amélioration est possible, la méthode renvoie `true` et fait les opérations nécessaires.

```
1  public bool Upgrade(ref int money)
2  {
3      // TODO
4  }
```

La troisième méthode à implémenter est `GetHousing`. Elle renvoie la la capacité en visiteurs générée par le bâtiment sur la tuile. Si la tuile ne possède pas de bâtiment ou si le bâtiment ne génère pas de "capacité en visiteurs", la fonction renvoie `0`.

```
1  public long GetHousing()
2  {
3      // TODO
4  }
```

De même pour `GetAttractiveness` et `GetIncome` qui renvoient respectivement l'attractivité et le revenu généré par la tuile.

```
1  public long GetAttractiveness()
2  {
3      // TODO
4  }
5
6  public long GetIncome(long population)
7  {
8      // TODO
9  }
```

3.2.4 Map

Il est temps d'implémenter la classe `Map`. Elle possède un seul attribut privé `matrix` qui est une matrice de `Tile`.

Implémentez son constructeur qui prend en paramètre une chaîne de caractère `name`, le nom de la carte. Vous **devez** utiliser la fonction `TycoonIO.ParseMap` qui prend le nom d'une carte en paramètre et renvoie une matrice de tuile.

```
1  public Map(string name)
2  {
3      // TODO
4  }
```

On veut pouvoir construire un bâtiment sur une tuile précise de notre carte pour cela implémentez la méthode `Build` qui prend en paramètre `i j` les coordonnées de la construction, `money` l'argent restant du parc et enfin `type` le type de bâtiment que l'on souhaite construire. La méthode renvoie `true` si le bâtiment est construit.

```
1  public bool Build(int i, int j, ref int money, Building.BuildingType type)
2  {
3      // TODO
4  }
```

Implémentez la fonction équivalente `Upgrade`.

```
1  public bool Upgrade(int i, int j, ref int money)
2  {
3      // TODO
4  }
```

Il faudrait pouvoir connaître le nombre de visiteurs dans le parc (la population). Pour cela, implémentez les trois fonctions suivantes :

`GetHousing` renvoie le nombre de visiteurs maximal que le parc peut accueillir.

```
1  public long GetHousing()
2  {
3      // TODO
4  }
```

`GetAttractiveness` renvoie le nombre de visiteurs qui souhaite visiter le parc.

```
1  public long GetAttractiveness()
2  {
3      // TODO
4  }
```

Enfin la fonction `GetPopulation` est le nombre réel de visiteurs. C'est tout simplement le minimum entre le nombre de personnes souhaitant visiter le parc et le nombre maximal de visiteurs que le parc peut accueillir.

```
1  public long GetPopulation()
2  {
3      // TODO
4  }
```

La dernière méthode à implémenter est `GetIncome` qui renvoie les revenus du parc.

```
1  public long GetIncome()
2  {
3      // TODO
4  }
```

3.2.5 Game

Il est temps d'implémenter la classe `Game`. Celle-ci possède les attributs suivants :
— `score` : de type `long` correspondant au score du joueur.
— `money` : de type `long` correspondant à l'argent restant du parc.
— `nbRound` : de type `int` correspondant au nombre total de tour.
— `round` : de type `int` correspondant au tour actuel.
— `map` : de type `Map` correspondant à la carte du jeu.

Tous ces attributs ne doivent pas être modifiables mais accessibles depuis une autre classe. Implémentez les **getters** pour tous ces attributs.

```
public long Score
{
    // TODO
}

public long Money
{
    // TODO
}

public int NbRound
{
    // TODO
}

public int Round
{
    // TODO
}

public Map Map
{
    // TODO
}
```

Implémentez le constructeur qui prend en paramètre le nom de la carte, le nombre de tour maximal et l'argent initial du joueur. Il est important que l'attribut `round` soit initialisé à `1`.

```
public Game(string name, int nbRound, long initialMoney)
{
    TycoonIO.GameInit(name, nbRound, initialMoney);
    // TODO
}
```

Note : La fonction `TycoonIO.GameInit` va vous permettre de tester le jeu. Il faut donc laisser cette ligne.

Vous allez devoir implémenter la méthode `Build` qui prend en paramètre une position et un type de bâtiment. Vous devez construire le bâtiment. La méthode renvoie `true` si l'opération à réussi. Si c'est le cas, vous devez également appeler la fonction `TycoonIO.GameBuild` qui prend en paramètre la position et le type du bâtiment construit.

```
public bool Build(int i, int j, Building.BuildingType type)
{
    // TODO
}
```

De même pour la méthode `Upgrade` qui améliore un bâtiment. En cas de réussite, vous devez appeler la fonction `TycoonIO.GameUpgrade` qui prend en paramètre la position du bâtiment.

```
1  public bool Upgrade(int i, int j)
2  {
3      // TODO
4  }
```

La méthode `Update` sera appelée à chaque fin de tour. Elle doit mettre à jour le score et l'argent restant. De plus, elle doit appeler la fonction `TycoonIO.GameUpdate`. Le score est la somme de l'argent gagné au cours de la partie.

```
1  public void Update()
2  {
3      // TODO
4  }
```

Pour finir, il vous faut la méthode `Launch` qui permet de lancer une partie. Celle ci prend en paramètre un `Bot`.

Un `Bot` est un objet qui possède au moins trois méthodes :
— `Start` : Cette méthode doit être appelée une seule fois au début de la partie.
— `Update` : Cette méthode doit être appelée à chaque tour. Elle va réaliser plusieurs actions. Par exemple, construire et améliorer un bâtiment.
— `End` : Cette méthode doit être appelée une seule fois à la fin de la partie.

Implémentez la méthode `Launch` qui va simuler `nbRound`. Elle doit appeler les différentes méthodes du bot au bon moment et le bon nombre de fois. De plus, elle doit maintenir la variable `round` à jour. Cette méthode renvoie le `score` final obtenu par le `bot`.

```
1  public long Launch(Bot bot)
2  {
3      // TODO
4  }
```

Conseil : Les méthodes du bot prennent en paramètre un objet de type `Game`. Il faut lui donner l'objet `game` en train de lancer le jeu. C'est-à-dire `this`.

3.3 Bot

Il est temps de faire votre premier `Bot`. Pour cela, vous disposez d'un exemple. Ouvrez le fichier`MyBot.cs`. Vous trouverez les trois méthodes obligatoires. On peut noter que la classe ne contient pas de constructeur, celui par défaut est donc utilisé.

3.3.1 Start

Cette méthode permet d'initialiser des variables propres à votre `Bot`. Dans l'exemple, nous n'avons aucune variable à initialiser, la fonction est donc vide.

```
1  public void Start(Game game)
2  {
3      // Nothing to do...
4  }
```

3.3.2 Update

La méthode la plus importante, c'est elle qui va construire et améliorer les bâtiments à chaque tour. Dans l'exemple, le bot va à chaque tour essayer de construire au même endroit une maison, une attraction et un magasin.

```
1 public void Update(Game game)
2 {
3     game.Build(2, 7, Building.BuildingType.HOUSE);
4     game.Build(12, 5, Building.BuildingType.ATTRACTION);
5     game.Build(8, 18, Building.BuildingType.SHOP);
6 }
```

3.3.3 End

Cette méthode peut vous permettre d'utiliser les résultats du bot et de l'améliorer en conséquence. Dans l'exemple, la méthode n'est pas utilisée.

```
1 public void End(Game game)
2 {
3     // Nothing to do...
4 }
```

3.3.4 Votre propre bot

Comme vous avez pu le remarquer, ce bot n'est pas très performant. Il ne prend pas en compte la carte du jeu et il ne réalise au mieux que trois actions. Vous devez donc implémenter un Bot qui réalise le plus grand score possible.

Votre bot sera testé sur différentes cartes. Pour tester votre code nous vous conseillons de créer d'autres cartes.

Nous vous conseillons aussi d'implémenter d'autre méthode le bot. Par exemple une méthode qui va sélectionner une tuile et construire un bâtiment dessus.
Vous pouvez aussi ajouter des fonctions aux différentes classes de manière à simplifier le code du bot. Par exemple dans la classe `Tuile` vous pouvez implémenter une méthode `IsBuildable` qui indique la possibilité de construire sur la tuile.

Attention : Vous ne devez pas modifier le comportement et les prototypes des méthodes décrites dans le sujet.

Vous devez commenter votre code et vos différentes fonctions. Vous pouvez décrire rapidement le fonctionnement de votre bot dans le `README`.

Vous disposez d'un programme qui permet de visualiser les actions de votre bot. Pour l'utiliser, il suffit d'appeler la fonction `TycoonIO.Viewer`.

Il est possible de soumettre les résultats de votre Bot. Pour cela, il suffit de soumettre votre TP sur l'intranet ACDC **(Attention à l'architecture)**. Nous vous rappelons qu'il est possible de faire autant de rendu que vous souhaitez.

Un tableau des scores est disponible ici : `...Soon...`
Le score affiché est le **meilleur** score réalisé par votre bot sur la carte *agave_plantation*. De plus si votre bot exécute une action invalide alors votre score sera de `0`.

Les meilleurs d'entre vous seront récompensés.

3.4 Bonus

Dans cette partie vous allez devoir implémenter l'action de destruction d'un bâtiment. De plus vous avez dû le remarquer, il est possible que le bot modifie directement la carte du jeu en modifiant les tuiles de celle-ci. Pour remédier à cela nous allons renvoyer une copie de la carte au bot au lieu de l'original. Cela permettra à vos bots de modifier la carte (ça copie).

3.4.1 Destroy

Nous allons commencer par implémenter la méthode `Destroy` de la classe `Tile`. Cette méthode renvoie `true` si la destruction du bâtiment est possible. Dans ce cas elle devra effectuer la destruction du bâtiment.

```
1 public bool Destroy()
2 {
3     // TODO
4 }
```

Rendez-vous dans la classe `Map`. Implémentez la fonction `Destroy` qui prend en paramètre la position du bâtiment à détruire. La méthode renvoie `true` si la destruction est possible.

```
1 public bool Destroy(int i, int j)
2 {
3     // TODO
4 }
```

Pour finir implémentez la méthode `Destroy` dans la classe `Game`. Cette méthode prend en paramètre la position du bâtiment à détruire et renvoie si l'opération à réussi. De plus si l'opération s'est effectuée elle devra appeler la fonction `TycoonIO.GameDestroy`.

```
1 public bool Destroy(int i, int j)
2 {
3     // TODO
4 }
```

3.4.2 Map copy

Pour pouvoir copier une carte entière il faut pouvoir copier tous les objets qui composent cette carte. C'est-à-dire `Tile` `Attraction` `Shop` et `House`. Vous allez devoir surcharger le constructeur de ces classes.

Implémentez les nouveaux constructeurs de ces classes.

```
public Attraction(Attraction attraction)
{
    // TODO
}

public Shop(Shop shop)
{
    // TODO
}

public House(House house)
{
    // TODO
}

public Tile(Tile tile)
{
    // TODO
}
```

Vous pouvez faire de même avec la classe `Map`. Ce constructeur devra créer une nouvelle carte en copiant tous les éléments de l'objet copié vers la copie.

```
public Map(Map map)
{
    // TODO
}
```

Pour finir il suffit de modifier le **getter** de la classe `Game` de manière à renvoyer une copie de la carte et non pas l'original.

```
public Map Map
{
    get
    {
        // TODO
    }
}
```

These violent deadlines have violent ends.