TP C#6: WestWorld Tycoon

Assignment

Archive

You must submit a zip file with the following architecture :

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

- You shall obviously replace *firstname.lastname* with your login (the format of which usually is *firstname.lastname*).
- The code **MUST** compile.
- In this practical, you are allowed to implement methods other than those specified. They will be considered bonuses. However, you must keep the archive's size as small as possible.

AUTHORS

This file must contain the following : a star (*), a space, your login and a newline.
Here is an example (where $ represents the newline and ␣ a blank space):

```
*␣firstname.lastname$
```

Please note that the filename is AUTHORS with **NO** extension. To create your `AUTHORS` file simply, you can type the following command in a terminal:

```
echo "* firstname.lastname" > AUTHORS
```

README

In this file, you can write any and all comments you might have about the practical, your work, or more generally about your strengths and weaknesses. You must list and explain all the bonuses you have implemented. An empty `README` file will be considered as an invalid archive (malus).

Contents

1 Introduction

1.1 Objectives

During this TP, we will go deeper on notions you have already seen in the TP C#4. We will ask you to code and understand objects interactions within a simple video game.
Here are the important notions presented in this TP:

- Class, Object
- Constructor and Destructor
- Static
- Public, Private
- Inheritance
- Overload and Override

2 Lesson

Objects Oriented Programming notions can be extremely complex. Especially interactions between them. That is why we don't ask you to fully understand every one of them. Some are only presented to trigger your curiosity. On the other hand, you will have to know the most important ones.

2.1 Reminder

Here are presented notions from previous practical C#4. You have to understand every one of them.

2.1.1 Classes and Objects

C# is an object oriented language. This means that it is possible to define and instantiate your own **objects**. But what is an **object** exactly?

An **object** is a data structure described by the **class** it is part of. Be aware, these two notions are not equivalent. **Class** and **object** can not be interchanged. The **class** describes the structure and the behaviour of the **object**. In C#, **classes** are used as a type as `int` or `float` is. The variable will then be considered as an **object**.

```
class MyClass //this is a class
{
}

public static void Main(string[] args)
{
  MyClass foo = new MyClass(); //this is an object
}
```

In this example you can see that the variable foo is an **object** of **class** MyClass. But there are many other way to describe the relationship between MyClass and foo.

- foo is an **instance** of MyClass.
- MyClass is the type of the variable foo.

2.1.2 Encapsulation

Classes are used to group and link information (**Attributes**) et behaviours (**Methods**). This is called *encapsulation*.

```
class MyClass
{
  int myInt; //this is an attribute
  string myString;
  float myFloat;

  void PrintMyString() //this is a method
  {
    Console.WriteLine(myString);
  }
}
```

Attributes describe information contained in the **object** and **methods** describe the way to interact with this information. Is is obviously possible to modify **attributes** of an **object** from a **method**.

2.1.3 Constructors et Destructor

Constructors and **destructor** are used to define lifespan of an **object**. An **object**'s birth is the call to its **constructor** and its death is the call to its **destructor**. A **constructor** does not have a return value and is always named after the **class** name.

```
class MyClass
{
  int myInt;

  public MyClass(int i) //this is a constructor
  {
    this.myInt = i;
  }
}
```

Constructors are often used to set **attributes**' default value to an **object**. The keyword `this` refers to the **object** itself and is used to distinguish between **attributes** and **method**'s parameters. You will not have to declare any **destructors**. C# is giving one by default (a default **constructor** is provided by C#). **Destructors** are used to clean up the environment (memory, etc) during the destruction of the **object**.

2.2 Visibility

You are most probably asking yourself what `static`, `public` and `private` mean. We are going to explain it today. These three notions must be understood.

2.2.1 Static

Let us start with the keyword `static`. It means that the following **attribute** or **method** does not depends of the **object** but of the the **object**'s **class** exclusively.

```
class Human
{
  static int globalPopulation = 0;
  Human()
  {
    ++globalPopulation;
  }
}
public static void Main(string[] args)
{
  var h1 = new Human();
  var h2 = new Human();

  Console.WriteLine(Human.globalPopulation) // globalPopulation = 2;
  //h1.globalPopulation      illegal instruction
}
```

As you can see, the **attribute** globalPopulation is not owned by an **object** any more. It is owned by the **class** Human instead. This is why it is not possible to access it through an **object** or the keyword `this`. You must use the **class** name directly.

A `static` **method** can not access nor modify `non-static` **attributes** of the **class**. Moreover, if a **class** is tagged as *static*, then every one of its **attributes** and **methods** must be tagged too. This is the case for the Program **class** for example.

2.2.2 public vs private

`public` and `private` keywords defines access constraints to **attributes** and **methods** of an **object**. `public` means that it is possible to access the **attribute/method** from outside the object with the **.** operator. (ex: `obj.attribute`).

`private` indicates that only the **object** itself can access its **attribute/method** from its other **methods**. If no keyword is specified, `private` is used by default.

```
class MyClass
{
  public string myPublicString;
  private string myPrivateString;

  public string myReadOnly
  {
    // Getter
    get { return myPrivateString;}
  }
  public string myWriteOnly
  {
    // Setter
    set { myPrivateString = value; }
  }
  public string myReadButNotWrite
  {
    get { return myPrivateString;}
    private set { myPrivateString = value; }
  }
}
```

`private` is important to guarantee some safety inside the source code program. It prevents you from manually modifying a string length for instance.

Attributes can also be defined as **read only** or **write only** thanks to a **getter/setter** system.

2.2.3 protected

There is another protection mode: `protected`. This keyword protects an **attribute/method** as the *private* keyword would except for one detail: The **attribute/method** is considered as *public* in any **sub-class** inheriting from the current class (inheritance will be explain in next part).

```
class MyClass
{
  private string myPrivateString;
  protected string myProtectedString;

  public MyClass(string myPrivateString)
  {
    this.myPrivateString = myPrivateString;
    this.myProtectedString = myPrivateString;
  }
}

class MySubClass : MyClass
{
  public MySubClass(string myPrivateString, string myProtectedString)
  : base(myPrivateString)
  {
      //this.myPrivateString = myPrivateString;   illegal instruction
      this.myProtectedString = myProtectedString;
  }
}
```

The `this` keyword is used to add safety to an application once again.

2.3 Inheritance

Inheritance is a key point in object oriented programming. It allows to prevent some code duplication but more over to share common roots between **classes**. The inheriting **class** is called **sub-class** and the inherited one is called **super-class**.

```
Item
|-- Equipment
|    |-- Armour
|    |-- Weapon
|    |-- ...
|-- Consumable
     |-- Food
     |-- ...
```

But inheritance is not limited to a simple genealogy. A second important point is that every **sub-class** can be used in place of any of its **super-classes** (this will become really useful for your S2 project). Here is a little example:

```
class Student
{
  protected int promotion;
  protected string name;

  public Student(int promotion, string name)
  {
    this.promotion = promotion;
    this.name = name;
  }

  public int GetPromo()
  {
    return this.promotion;
  }
}

class Sup : Student
{
  public bool sharp;

  public Sup(int promotion, string fullname, bool isSharp)
  {
    this.promotion = promotion;
    name = fullname;
    this.sharp = isSharp;
  }
}

class Ing1 : Student
{
  public bool assistant;

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

As you can see, every **sub-class** can access its **super-class** (*public* and *protected*) **attributes** and **methods** directly (with or without `this`). The `base` keyword refers to the equivalent **method** of its **super-class** (we will come back to it latter).

```csharp
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; illegal instruction
10   //a1.assistant; illegal instruction
11 }
```

Be aware, it is possible to implicitly convert a **sub-class** into a `super-class` but not the other way around. This is why line 5 is correct whereas line 10 is not. Even tough `a1` was instantiated as an `Ing1`, its declaration as a `Student` does not grant it access to the Ing1 **class attributes**. Whether an **object** is part of a class or not with the keyword `is`.

```csharp
public static void Main(string[] args)
{
  Student a1 = new Ing1("Florian Amsallem", 2020, true, false);
  if (a1 is Ing1) // true here
    // do something
}
```

2.3.1 abstract and sealed

It is possible for a **class** to never want to be **instantiated**. It will only serve as a common structural definition for other **classes** inheritance. The keyword *abstract* prevents any **object** from being created from such a **class**.
It is also possible to declare **classes** that should not be inherited from. In this case, the *sealed* keyword should be used.

```csharp
abstract class MCQ<T>
{
  protected string question;
  protected int answer;
  protected T[] choices;
  public bool IsCorrect(int choice)
  {
    return choice == answer;
  }

  public void AskQuestion()
  {
    Console.WriteLine(question);
    foreach (T choice in choices)
      Console.WriteLine(choice);
  }
}

sealed class MathMCQ : MCQ<int> { }
sealed class FrenchMCQ : MCQ<string> { }
```

This little example is here to showcase the use of the `abstract` and `sealed` keywords, as well as to be your first contact with **generics** (`Class<Type>`). It is really not necessary that you understand how **generics** work at this point.

The `abstract` and `sealed` can also be used on a unique **attribute** or **method**.

2.4 Polymorphism

Polymorphism is the last key point about object oriented programming. It means that an **object**, **method** or **attribute** can change form during execution. We already talked a bit about it without you noticing it. The fact that an **object** can use the type of its **super-class** is **polymorphism**. It can seem quite weak at first glance because nearly every **method** and **attribute** of a **super-class** are accessible within the **sub-class**.

Going back in the genealogy tree of a **class** would mean losing access to **attributes** and **methods** introduced by **sub-classes**. That is completely true, but a very simplistic view since it is possible to redefine a **method** during inheritance. In that case, polymorphism allows accessing the initial version of a **method**.

It is important that you understand the following notions.

2.4.1 Overload

Let us start simple with **overloading**. It allows using the same name for two or more functions/methods. There are still is some constraint:

- All function of the same name must have different **signatures**. Meaning that the combination of the return type, the function name and the parameters type (by order of arrival) must be unique;
- A change in return type is not enough (at least one parameter must be unique).

No need to prefix or suffix your recursive functions anymore. Overloading is also used to create multiple **constructors** for a **class**.

```
public int sum(int[] arr, int pos)
{
  if (pos < arr.Length)
    return arr[pos] + sum(arr, pos + 1);
  return 0;
}

public int sum(int[] arr)
{
  return sum(arr, 0);
}
```

```
public class vector2
{
  public int x;
  public int y;
}

public class Point
{
  private vector2 pos;

  public Point(Point p)
  {
    Point(p.pos);
  }

  public Point(vector2 vec)
  {
    Point(vec.x, vec.y);
  }

  public Point(int x, int y)
  {
    this.pos.x = x;
    this.pos.y = y;
  }
}
```

2.4.2 Substitution

Substitution consists in changing a **method** or **attribute** definition. There are two ways to do it :

- Overwriting the previous version with `new`.
- Adapting it with `override`.

But only **methods** tagged as `virtual`, `override` or `abstract` can be redefined with `override` in a **sub-class** whereas `new` can overwrite everything.

Be aware, every **method** tagged with *abstract* must be **overridden** in every **sub-classe**.

```csharp
class Student
{
  protected int promotion;
  protected string name;

  public Student(int promotion, string name)
  {
    this.promotion = promotion;
    this.name = name;
  }

  public int GetPromo()
  {
    return this.promotion;
  }

  public virtual void SayHi()
  {
    Console.WriteLine("I'm " + name);
  }
}
```

```csharp
class Sup : Student
{
  public bool sharp;

  public Sup(int promotion, string fullname, bool isSharp)
  {
    this.promotion = promotion;
    name = fullname;
    this.sharp = isSharp;
  }

  public override void SayHi()
  {
    base();  // Console.WriteLine("I'm " + name);
    if (sharp)
      Console.WriteLine("And I'm in sharp");
  }
}
```

Inside of **overridden methods**, it is possible to access the **super-class method** using the `base` keyword. This keyword is usually used in **constructors**.

```
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;
  }
}
```

These mechanisms allow you for having multiple functions with the exact same **signature** but different behaviours.

But keep in mind that it is not possible to access every definition of a **method** from an **object** directly. The definition chosen by the compiler depends on the **object**'s current type. That is why it is possible, thanks to type conversion, to choose the appropriate definition.

```
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(); // convert c1 into Student, Then call SayHi.
  /*
  I'm Julien Mounier
  */
}
```

Be aware, overridden **methods** can't lose but only gain in protection level. It is possible to `override` from `public` to `private` but not the opposite. `protected` is located between the two.

3 Exercise : WestWorld Tycoon

3.1 Introduction

Congratulation! You are candidate to become the manager of **WestWorld**! To select the best out of you all, we prepared a test. You'll have to manage a virtual amusement park: *WestWorld Tycoon*.

WestWorld Tycoon is a management simulation game of an amusement park. In this game, you can build buildings like shops, houses and attractions obviously. Your goal is to earn as much money as possible within a limited amount of time. The candidate who earned the most will be enrolled as manager of the real park.

3.1.1 Objectives

In this exercise, you will have to implement the game *WestWorld Tycoon*. You will also have to implement a program to automatically manage the park as good as possible (a bot). To help you, we give you data structure that you can download from the Intranet. In order to get a mark, you must follow the structures.

In general, we wish that your bot can not cheat. For example, the bot shouldn't be able to modify remaining money.

3.1.2 Game description

The park is represented by a matrix (the map). The map is divided into three types of biomes:

- *Sea*: a field of water that prevent the construction of any building.
- *Mountain*: mountain that blocks the construction of buildings.
- *Plain*: a place where you can build anything.

The bot can do any of this three actions:

- *Build*: Only on an empty plain.
- *Upgrade*: any building can be upgraded to improve its effects.
- *Destroy*: every building can be destroyed for free.

There is three types of buildings:

- *Attraction*: Improve the population of the park.
- *House*: Improve the maximal population capacity of the park.
- *Shop*: Improve the income generated by the park.

3.2 Rules

This part is necessary for you to start the next one. Its goal is to implement the rules of the game.

3.2.1 Attraction

Open the file `Attraction.cs` ! There is two things to see:

- The class `Attraction` inherit from the class `Building` which is already implemented.
- There is two constants in the class `Attraction`. The building and the upgrading price. Do not modify these values.

The first thing you have to do is to add an **attribute** `lvl` to the class. `lvl` is of type `int` and must be **private** to prevent the level of the attraction to be modified from outside of the class.

You can now implement the class constructor. It should initialise every attributes (do not forget the ones from the super-class).

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

You must now add a **getter** to grant a read-only access to the attraction's level.

```
public int Lvl
{
    //TODO
}
```

You only have to implement the `Upgrade` and `Attractiveness` methods.

The `Upgrade` method improve the level of the attraction. It must take a reference to an integer `money` which corresponds to the remaining money in of the park. The method must return `true` if the upgrade is possible. It must also upgrade the attraction if possible (subtract the cost to `money` and increment the level).

```
public bool Upgrade(ref int money)
{
    // TODO
}
```

Hint: You must use the constant array `UPGRADE_COST` declared previously. As its name suggest it, this array contains the upgrading cost. It contains only three values because an attraction can not be upgraded more than three time.

Finally, you will implement the method `Attractiveness` which returns a `long` corresponding to the attractiveness generated by the building. This value depends on the attraction level.

```
public long Attractiveness()
{
    // TODO
}
```

Hint: You must use the constant array `ATTRACTIVENESS`. For example, at level 0, an attraction is generating `ATTRACTIVENESS[0]` attractiveness.

3.2.2 Shop & House

You can now reproduce the exact same process for the `Shop` and `House` classes.

```
public Shop()
{
    // TODO
}

public int Lvl
{
    // TODO
}

public long Income(long population)
{
    // TODO
}

public bool Upgrade(ref int money)
{
    // TODO
}
```

The revenue given by `Income` depends on the shop level. The constant array `INCOME` gives you the percentile of the population that buys in this shop. We will consider that any goods sells for 1 dollar.

```
public House()
{
    // TODO
}

public int Lvl
{
    // TODO
}


public long Housing()
{
    // TODO
}

public bool Upgrade(ref int money)
{
    // TODO
}
```

The housing capacity `Housing` depends on the house level. This value is given in the `HOUSING` array. For example, a level 0 house generates `HOUSING[0]` visitors.

3.2.3 Tile

We will now consider the tiles of our map (the class *tile*).

A tile has two attributes:

- *biome*
- *building* which value is `null` if no building is present on the tile.

Add this two attributes to the class `Tile`. This attributes are `private`.

You must now create the constructor which takes `Biome` as parameter. The building attribute must be set to `null`.

```
public Tile(Biome b)
{
    // TODO
}
```

You must now add a **getter** to grant a read-only access to the tile's biome.

```
public Biome GetBiome
{
    // TODO
}
```

You can now implement the class methods. The first one, `Build`, takes the reference to an integer `money` which is the remaining money of the park. It also takes `type`: the building type. This method must return `true` if the construction is possible and create the building. You should update the remaining money. Do not forget to consider the biome of the tile.

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

The second one is `Upgrade`. This method only takes a reference to the integer `money`. If the upgrade is possible, the method should return true and update the level of the building as well as the money.

```
public bool Upgrade(ref int money)
{
    // TODO
}
```

The third method to implement is `GetHousing`. It must return the number of visitors generated by the building on the tile. If there is no building or the building on the tile does not produce any housing, the function must return `0`.

```
public long GetHousing()
{
    // TODO
}
```

Do the same for GetAttractiveness and GetIncome which return respectively the attractiveness and the income generated by the tile.

```
public long GetAttractiveness()
{
    // TODO
}

public long GetIncome(long population)
{
    // TODO
}
```

3.2.4 Map

It is now time to implement the class Map. There is only one public attribute matrix, a matrix of tiles (Tile).

Implement its constructor which takes a string name (the name of the map) as parameter. You **must** use the function TycoonIO.ParseMap which takes a string and returns a matrix of tiles.

```
public Map(string name)
{
    // TODO
}
```

We now want to build on tiles of the map. In order to do it, you have to implement the method Build which takes i j the position of the tile, money the money of the park and at last type the type of the building we wish to build. This method returns true if the building has been build.

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

Implement the equivalent method but for Upgrade.

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

It would be nice to now the number of visitors (population) of the park at any given time. To do that, you will implement three functions:

GetHousing return the maximum number of visitor that the park can host.

```
public long GetHousing()
{
    // TODO
}
```

GetAttractiveness return the number of visitors willing to come to the park.

```
public long GetAttractiveness()
{
    // TODO
}
```

And finally, GetPopulation which return the real number of visitor in the park. Simply the minimum between the people willing to come and the maximum your park can host.

```
public long GetPopulation()
{
    // TODO
}
```

The last method to implement is GetIncome which returns the revenue of the park.

```
public long GetIncome()
{
    // TODO
}
```

3.2.5 Game

It is time to implement the class Game, possessing the following attributes:

- score: type long your score.
- money: type long remaining money in the bank.
- nbRound: type int total number of rounds before the end.
- round: type int number of round since the start of the game.
- map: type Map the map.

All these attributes must be read-only. Implement the corresponding **getters** for all these attributes.

```csharp
public long Score
{
    // TODO
}

public long Money
{
    // TODO
}

public int NbRound
{
    // TODO
}

public int Round
{
    // TODO
}

public Map Map
{
    // TODO
}
```

Implement the constructor which takes the name of the map, the maximum number of rounds and the starting money. The game must start at `round 1` .

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

NB: The function `TycoonIO.GameInit` will allow you to test your game. So leave the line where it is.

You will have to implement the method `Build`. It takes a position and a building type. The method must build the building and return `true` on success. On success you must also call the function `TycoonIO.GameBuild` which takes the same parameters as `Build`.

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

Do the same for the function `Upgrade`. On success, you should call the function `TycoonIO.GameUpgrade`.

```csharp
public bool Upgrade(int i, int j)
{
    // TODO
}
```

The method `Update` will be called at the end of every round. It must update the score and the remaining money. But most importantly, it must call the function `TycoonIO.GameUpdate`. Th e score is the sum of every buck you have earned since the start of the game.

```
public void Update()
{
    // TODO
}
```

Finally, you will need the function `Launch` which lunch a game with a `bot` given in parameter. A `Bot` is an object which have at least the following three methods:

- `Start`: method that initialize your bot and will only be called at the start of the game.
- `Update`: This method is called during every round once. This in this method that your `bot` will chose to do any number of actions (build, upgrade, nothing, etc).
- `End`: this method must be called at the end of the game.

Implement the method `Launch` which runs `nbRound`. This method must call the right function of the bot at the right time. It must also maintain the different attributes value and return the final score of the bot in the end.

```
public long Launch(Bot bot)
{
    // TODO
}
```

Hint: The bot's methods take a variable of type `Game` as parameter. You must give it the current object `game` object (`this`).

3.3 Bot

It is time for you to create your first `Bot` from scratch. Do not worry, we gave you a simple bot to find inspiration. Open the file `MyBot.cs`. You can find the three mandatory methods. You can see that there is no constructor. It means that the default one is used.

3.3.1 Start

This method allows you to initialise your `Bot` own attributes. In the example, we do not use any variable. So we do not have to initialize any variable and the function is empty.

```
public void Start(Game game)
{
    // Nothing to do...
}
```

3.3.2 Update

The most important method, the one which will build and upgrade your park and be called in every round. In the example, the bot will try to construct a house an attraction and a shop on the same respective locations.

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

3.3.3 End

This method can allow you to check the bot's results and improve it accordingly. In the example the method is not used.

```
public void End(Game game)
{
    // Nothing to do...
}
```

3.3.4 You Own Bot

As you have seen, this bot is not really efficient. It does not take into account the map of the game achive only three actions maximum. You should implement a bot that scores as much as possible.

We greatly advise you to add your own methods to your bot. One to select a tile and build on it for example.
You can also add functions to the other classes you have previously implemented if it simplify the work of your bot. You can add one to the class `Tile` to tell is the tile is build-able or not.

Warning: You must not modify the behaviour and the signature of any of the methods described in the subject in any way what so ever.

You must also document your code and different functions. Just describe what your function do. You can describe what your bot do in your `README`.

There is a program at your disposal which will show you the actions taken by your bot. To use it, you just have to call the function `TycoonIO.Viewer`.

It's possible to submit the results of your Bot. To do this, simply submit your assignement on `https://acdc.cri.epita.net/` **(Pay attention to the architecture)**. We remind you that you can submit your assignement as many times as you want.

The scoreboard is available here: `...Soon....`

The displayed score is the highest score your bot did on the *agave_plantation* map. Invalid actions will give you a score of `0`.

The bests managers will be awarded.

3.4 Bonus

In this part you will have to implement the destruction of a building. As you might have seen, it is possible for your bot to modify the tiles of a map. To keep the past of the game and your bot simulation synchronous, you will have to create clones of the map before starting computation. Your bot will then be allowed to modify the copied map without any risk of breaking any rule.

3.4.1 Destroy

We will start with the destruction be implementing the method `Destroy` of the class `Tile`. This method return `true` and destroy the building if is destruction is possible.

```
public bool Destroy()
{
    // TODO
}
```

Let us go to the class `Map`. Implement the method `Destroy`. It takes the position of the building to destroy.

```
public bool Destroy(int i, int j)
{
    // TODO
}
```

At last, implement the function `Destroy` in the `Game` class.Do not forget to call the function `TycoonIO.GameDestroy` on success.

```
public bool Destroy(int i, int j)
{
    // TODO
}
```

3.4.2 Map copy

To copy an entire map you must be able to copy every of its component. (aka: `Tile`, `Attraction`, `Shop` and `House`). You will have to add an overload to these class constructors.

Here are the function signatures.

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

public Shop(Shop shop)
{
    // TODO
}

public House(House house)
{
    // TODO
}

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

You can now do the same with the `Map` constructor. Do not forget to copy every component.

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

Finally, you just have to modify the map **getter** in the `Game` class so that it returns a copy of the map and not the original.

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

These violent deadlines have violent ends.