TP C#8
Tiny SimCity

Submission

Archive

You must submit a .zip file respecting the following architecture:

```
rendu-tp-firstname.lastname.zip
    |-- rendu-tp-firstname.lastname/
        |-- AUTHORS
        |-- README
        |-- MySimCity
            |-- MySimCity.sln
            |-- MySimCity
                |-- Building.cs
                |-- CityHall.cs
                |-- Factory.cs
                |-- House.cs
                |-- PoliceStation.cs
                |-- Program.cs
                |-- Shop.cs
                    |-- Everything except bin/ and obj/
```

- Obviously, *firstname.lastname* must be replaced with your login.
- Your code must compile and be readable.
- For this subject, the architecture can be changed depending on the bonuses you do. But be sure to write your changes in the *README*.

AUTHORS

The *AUTHORS* file must contain: a star (*), a space, your login, and finally a new line as in the example below ($ represents a new line and ␣ a space):

```
*␣firstname.lastname$
```

The *AUTHORS* file has no extension. In order to create a valid *AUTHORS* file, you can open a terminal and enter the command below:

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

README

You must write any commentary, extra work explanations, feedback in this file. An empty *README* will be considered as incorrect. Like AUTHORS, this file must have no extension.

1 Introduction

1.1 Presentation

During the past few weeks, you have been taught how to use the C# language enough to be able to start working on more "complex" projects. That is what we are going to see this week. You will be brought to use the following concepts:

- The console
- C# classes
- The debugger (We strongly recommend you to)
- Event handlers

If you are not feeling confident enough with these concepts, you can still refer to the previous projects or have a look at what should be your bible[1].

This week, you will have to work on a small SimCity. The whole exercise will be guided, and we will suggest steps to help you work on this project. Furthermore, we will allow you to show us what you can do through a free bonus part. You are free to add whatever you want to improve your game. Use your imagination and don't forget to write the changes you've made in the *README*.

1.2 Reminder

Abstract classes

During the previous projects, you had to use classes that are already implemented so as to make your work easier (Console). You also created your own classes and your own class methods. In this project, we are going to see what abstract classes are. We will see why they are useful to us and why we want to use them in this project.

[1] MSDN: `https://msdn.microsoft.com/`

One of the main concepts of oriented object programming is to be able to organize and represent your code in a way that is easily understood from our point of view. At some point, we will manipulate several classes with common attributes within a single data structure. In this project, we will create different kinds of building. For instance, we will have houses, police stations, a city hall, etc. These buildings will be represented by different classes. The main challenge will appear when we will try to manipulate these buildings within a single list without knowing their types. C# is a strongly-typed language, and thus doesn't allow us to put objects from different classes in the same data structure. A (very) dirty way to fix this problem would be to create a new list for each type of building we have in the project. Your code would then be duplicated and be less maintainable. Here, we will use an abstract class **Building** to be able to "wrap" every building in our game in order to store them in a single structure.

Here is an example of an abstract class:

```
1 abstract class Building
2 {
3     protected int cost;
4     public abstract void Print();
5 }
```

You will notice the appearance of the **abstract** keyword which is added not only to the class definition, but to its method prototypes. This class being an abstract class means it cannot be instantiated. Every function in the class must be written as abstract. This concept will be explained in this project's coding part.

Inheritance

Inheritance is an object oriented programming concept which completes the use of our abstract class in this project. Indeed, our **Building** class is not instantiable and we would like to create our own buildings. To do so, we are going to create several classes that will inherit the **Building** class. An **A** class that extends a **B** class inherits every attribute and method from the **B** class. We have two main benefits from using inheritance: avoiding code duplication and allowing the manipulation of objects that are instances of different classes as objects from the class they inherit. For example, if a **House** class inherits from a **Building** class, then we can manipulate an object instantiated from the **House** class as an object from the **Building** class. Thus if we have another class called **Hospital** that inherits from **Building**, we can create a list containing **Building** objects to store both hospitals and houses.

Here is an example of inheritance:

```
class House: Building
{
    private bool occupied;
    private int inhabitants;
    public House()
    {
        // Constructor ...
    }
    public override void Print()
    {
        // Code ...
    }
}
```

The **House** class inherits from the **Building** class. Thus, it inherits every attribute and method as well. In this example, the **Building** class has a **cost** attribute and a **Print()** method. You will notice that when a class inherits from an abstract class, you must provide an implementation for each abstract method in the abstract class. That's what will allow us to call the abstract class functions without knowing the type of the object beforehand. Here, we have to define the method **Print()** by adding the keyword **override** to the method's prototype.

Properties

You were taught how to declare your class attribute with a private visibility and access it outside the class thanks to getters and setters. These functions allow you to control how your users will access your private attributes. However, having to write two methods for a single private attribute can sometimes become heavy, especially when your classes will contain a lot of attributes. Besides, you'd prefer to access an attribute by its name rather than the name of a method (such as **GetAttribute()**.

Here is an example of a property:

```
class MyClass
{
    private int myInt;
    public int MyProperty
    {
        get { return this.myInt; } // Getter
        set { this.myInt = value; } // Setter
    }
}
```

Let's have a look at the syntax:

> **<access-modifier> <type> <name> { <accessors> }**

First, we want our property to be visible from outside the class so we can access it. Thus, we are putting it's visibility to **public**. Then, as for a variable, you can see there is a type and a name. Finally, the most important, the accessors. Property accessors are defined thanks to two keywords: **get** and **set** which allow us to control how the attribute will be accessed and how it will be set, respectively. What you should know is that a property is just some sort of interface in order to define how an attribute is accessed. Thus an attribute must be declared beforehand. We will set this attributes as private so that we cannot access it directly, and thus force us to use the property. Here, we declared **`MyProperty`** in order to access the attribute **`myInt`**.

We can use the property like this:

```
1 public static void Main(string[] args)
2 {
3     MyClass myInstance = new MyClass(); // Create a new object
4     int x = myInstance.MyProperty; // x = 0
5     myInstance.MyProperty = 666; // myInt = 666
6 }
```

We can rename **`MyProperty`** and give it a name that makes it easier to understand. Let's comment this code. At line 3, we just create a new object with the default constructor. Therefore, **`myInt`** is initialized to 0. Then, we want to access **`myInt`** using the property in order to read its value and store it in a variable. This access is defined in the code close to the **`get`** keyword. Finally, we use our property to access the object's attribute but to assign a new value to it (666). This new value corresponds to the **`value`** keyword in the **`set`** definition which contains instruction regarding how the attribute will be set.

We still have a problem. The definition of a property remains too long just to define "normal" accesses (without specific behaviors) like we just did. In C#, there are what we call **auto-implemented properties** in order to help use reduce the amount of code written. Thanks to them, the whole **`MyClass`** definition can be re-written as follow:

```
1 class MyClass
2 {
3     public int MyProperty { get; set; }
4 }
```

And that's all. *Wait a second. I don't see where the attribute is. I thought properties were just a way to access our attributes and are not fields themselves.* That is correct, but with auto-implemented properties, a backing field is declared for you. We advise you to use properties throughout this project instead of accessing your attributes with methods.

Do not hesitate to read this part several times or ask your ACDC if you do not understand these concepts.

2 Basics

Now it is time for you to start coding your SimCity. Be assured that what you will accomplish during this project will be far from a real SimCity once finished. Still, the bonus part will allow you to impress us since you will be able to improve your SimCity to make it look like a real one. We strongly advise you to read the entire subject before starting. You will have a better understanding of the subject and will know beforehand how you are going to put the pieces together. When we took a look back at the previous projects, we thought that you were not trained enough in house designing. Don't worry, this project will be your masterpiece.

2.1 Buildings

First of all, we need to create a class that will represent a building in our game. However, we will not create any object representing a *building*, but a more specific one such as a house, a factory, etc. Therefore, you must create an abstract class **`Building`** which will be used as a starting point for every other building class we will create in this project. We advise (require) you to write your classes in different files. To add a new file, go into your project's solution view (Alt+Shift+S if you can't see it), right click on your project's folder then *Add* > *New File* > *General* > *Empty Class*. We do not want to see 5 classes per file. Be sure to respect the project's architecture.

```
1 abstract class Building
2 {
3     // Fix me ...
4     public abstract void Print();
5 }
```

In the same file, declare an enumeration containing all the kinds of building that can be possibly built in your game. To start we will just use the following values : **`House`**, **`CityHall`**, **`PoliceStation`**, **`Factory`**, **`Shop`**. You can add more values for bonuses. This class should contain the abstract method **`Print()`** which you will need to implement in every derived class. It will also contain an attribute to store the type of building (enum).

2.2 A dictator

As a mayor, you must have a HQ you will be proud to call "Home". Thus, you need to create a **`CityHall`** class which will contain the main elements of your city hall. **`CityHall`** inherits from **`Building`**.

```
class CityHall : Building
{
    // Fix me ...
}
```

Don't forget to implement the abstract methods inherited from the abstract class, otherwise your code will not compile. Implement a constructor so as to initialize the class variables.

You can print your city hall any way you want[2]. But be reasonable when it comes to the dimensions of your city hall in order not to ruin your output.

A city hall:

```
/-----------------------\
|      [ CITYHALL ]     |
\-----------------------/
   | |   | |   | |   | |
   | |   | |   | |   | |
   | |   | |   | |   | |
   | |   | |   | |   | |
   | |   | |   | |   | |
   | |   | |   | |   | |
|-----------------------|
|-----------------------|
|_______________________|
```

2.3 Slaves

Now that you have your palace, you need a people and a way to host them. Create a **House** class that will allow you to do so. This class inherits from the **Building** class.

```
class House : Building
{
    // Fix me ...
}
```

Don't forget to implement the abstract methods inherited from the abstract class, otherwise your code will not compile. Implement a constructor so as to initialize the class variables.

[2]The two of us aren't pretty good (very bad) in ASCII design.

A house:

```
/----------\
|            |
| [ ]   [ ]  |
|            |
|____________|
```

Each house will be able to host 30 inhabitants at most. If there aren't enough houses, your town's population will not be able to increase.

2.4 Peacekeepers

The people can get out of control! With all their demonstrations, strikes, the terrorists, the crime and the political opponents, you need some manpower in order to maintain order and discipline in your town.

Implement a **`PoliceStation`** class which will obviously represent a police precinct. This class inherits from the **`Building`** class.

```
1 class PoliceStation : Building
2 {
3     // Fix me ...
4 }
```

Don't forget to implement the abstract methods inherited from the abstract class, otherwise your code will not compile. Implement a constructor so as to initialize the class variables.

A police precinct:

```
/------------\
|              |
| [ POLICE ]  |----------\
|              |  _______  |
|              | |------| |
|______________|_|______|_|
```

2.5 Work you slackers!

Your people could fall into eternal boredom. Fortunately, you are here to help them.

Create a **Factory** class that will allow you to give work to your people.

```
class Factory : Building
{
    // Fix me ...
}
```

Don't forget to implement the abstract methods inherited from the abstract class, otherwise your code will not compile. Implement a constructor so as to initialize the class variables.

A factory:

```
  ||
  ||
  ||  |\  |\  |\
  ||_|  \|  \|  \
|                 |
|                 |
|                 |
|_________________|
```

Each factory will be able to hire at most 100 inhabitants. If there is not enough work, the population won't be able to increase.

2.6 Consumer society

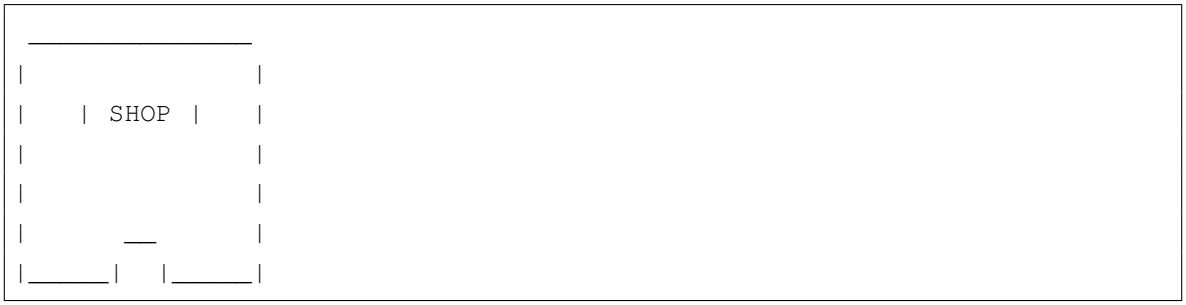
Your people now earn money from their work so they can spend it in local shops.

Create a **Shop** class that will represent a shop in your town. This class inherits from the **Building** class.

```
class Shop : Building
{
    // Fix me ...
}
```

Don't forget to implement the abstract methods inherited from the abstract class, otherwise your code will not compile. Implement a constructor so as to initialize the class variables.

A shop:



Each shop will offer 5 available jobs to the city and will be able to sell its goods to 50 inhabitants at most. If there aren't enough shops, the population won't be able to increase.

3 Build your town

Since you have all the buildings you need, you can now create your town.

Create a **Town** class.

```
class Town
{
        // Fix me ...
}
```

This class must contain three methods: one to be able to build a new building, another to destroy a building and a last one to print them all.

```
public bool AddBuilding();
public bool DestroyBuilding();
public void PrintBuildings();
```

The first two methods return **true** if the building was successfully built/destroyed. You must implement the methods **PrintState()** that will print your town's current state in the console (your current money, number of buildings and the population), as well as the **Update()** method which will update these values. The **Update()** method is the center of the game. It is here that you will code every change in your town depending on its current state. For instance, your money will change depending on the number of shops and the number of police stations in your town. This method will be called several times in your **Main** method in order to update the state of your city.

```
public void PrintState();
public void Update();
```

Example of PrintState():

```
Current money: 9000
Number of buildings: 12
Population: 666
```

4 Main method

Now, you must write your main method so as to finalize your game. In order to help you out, we regrouped every rule of the game that you must implement:

- The game starts with a fixed amount of money and an empty town. You are free to put the values you want.
- First, you must build your city hall. There can be only one city hall in your entire town and you cannot create other buildings without a city hall.
- The state of the town will be printed on the console by default.
- The player will have three choices: build something, destroy something and spend some time.
- Building something requires money. Obviously, if the player doesn't have enough money, then the building cannot be built.
- Destroying a building gives the player a part of the building's price back.
- Spending some time means printing the state of the city continuously without asking for an action. The state of the city must be printed each time it changes. To exit the waiting state, the player has to press any key.

Here are the rules related to the buildings:

- **CityHall**: the main building in your town. It is unique and cannot be destroyed. It doesn't have any special feature.
- **Factory**: allows you to create 100 jobs for your city. If you don't have enough jobs to offer, then your town's population will not increase.
- **House**: allows you to host 30 inhabitants at most. The population cannot increase if there are not enough houses in the player's town.
- **PoliceStation**: allows you to limit the crime rate in your city. This crime rate increases as your population grows. We assume that 30% of your population are criminals[3]. When the crime rate reaches 15%, the population of your town will not grow. A police station creates 20 jobs in your town, but requires money to maintain each time the **Update()** method is called.

[3]It really is a dangerous city

- **Shop**: allows you to offer your people products to be able to survive. You will also take this opportunity to apply outrageous taxes on these products. A shop offers 5 jobs to the city and earns money each time the **Update()** method is called.

To make it simple, your **Main** method will look like a huge loop where you will print your game's menu so the player can select an action and can to make the necessary changes to your town. Therefore, you must create a town beforehand so as to be able to call its methods to build or destroy a building.

It was said that the **Update** method would be called several times. We recommend that you create a timer[4] where you will call your method every X seconds. You are free to choose the time it takes for your town to update as long as it is below 10 seconds.

5 Bonus

This section is free. This means that you are allowed (and advised) to code any bonus you want as long as it improves the game's quality. Maybe you can make it look like a real SimCity. All of these bonuses must be explained in the README. Here are some examples of what you can do.

5.1 BURN!!!

Let us remind you that this game is supposed to be a SimCity, and like in any SimCity you are a tyrant that spends his time punishing your people for no reason. You will have to implement natural disasters in order to destroy random buildings and kill a large number of citizens. These slackers shouldn't have demonstrated for higher wages.

5.2 Printing

When we play a SimCity-like game we can choose where a building should be built. Thus, you must print your town seen from above. You must be able to chose where you want to build something with your arrow keys. Obviously, you cannot build something above an already existing building.

The code is the law.

[4] System.Timers