# TP C#8
# Mini SimCity

## Rendu

### Archive

Vous devez rendre un fichier au format .zip respectant l'architecture suivante:

```
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
                |-- Tout le reste sauf bin/ et obj/
```

- Bien entendu, *firstname.lastname* correspond à votre login et vous devez le remplacer par votre prénom et votre nom.
- Votre code doit compiler et être lisible.
- Exceptionnellement, l'architecture de rendu peut être modifié dans le but d'ajouter les bonus de votre choix. Par exemple, si vous ajouter une classe **`Hospital`**, vous pouvez ajouter un autre fichier .cs à votre archive.

### AUTHORS

Ce fichier doit contenir une ligne formatée comme il suit : une étoile (*), un espace, votre login et un retour à la ligne, RIEN DE PLUS.
Voici un exemple (où $ représente un retour à la ligne et ␣ un espace):

```
*␣firstname.lastname$
```

Notez que le nom du fichier est AUTHORS avec AUCUNE extension. Pour créer simplement un fichier AUTHORS valide, vous pouvez taper la commande suivante dans un terminal:

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

### README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses. Un README vide sera considéré comme une archive invalide (malus).

# 1 Introduction

## 1.1 Présentation

Durant les semaines précédentes, vous avez appris à utiliser suffisamment de fonctionnalités propres au C# pour pouvoir commencer à faire des petits projets plus amusants, plus concrets. C'est ce que nous allons voir dans le TP de cette semaine. Lors de cette séance, vous serez amenés à utiliser les notions suivantes:

- La console
- Les classes
- Le debugger
- Les événements

Si vous n'êtes pas encore à l'aise avec ces notions, vous pouvez toujours relire les précédents travaux ou bien consulter ce qui devrait vous servir de bible[1].

Cette semaine vous allez devoir réaliser le projet *Mini SimCity*. L'exercice sera guidé et proposera des étapes afin de vous aider à réaliser ce projet. De plus, ce TP va vous permettre de nous montrer ce que vous pouvez faire puisqu'une partie de l'implémentation de votre jeu vous sera laissée libre. A vous

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

d'utiliser votre imagination afin de compléter votre TP avec des bonus qui amélioreront la qualité de votre rendu.

## 1.2 Cours et rappels

### Les Classes Abstraites

Lors des travaux précédents, vous avez pu manipuler des classes qui existent déjà et qui sont présentes pour vous faciliter certaines tâches (Console). Vous avez également pu créer vos propres classes, ainsi que vos propres méthodes de classe. Ce que nous allons voir ce sont les classes abstraites, à quoi elles peuvent servir et pourquoi nous allons vouloir les utiliser dans ce TP.

Un des principes de la programmation orientée objet est de pouvoir organiser notre code, et de le représenter sous une forme qui nous est logique. Vous verrez rapidement que nous sommes amenés à avoir plusieurs objets issus de classes différentes, ayant des points communs d'un point de vue logique, et qui devront être manipulés au sein d'une même structure. Dans le cas de ce TP, nous allons devoir créer des bâtiments de différents types. On aura par exemple des maisons, des postes de police, etc. Ces bâtiments sont représentés par des classes différentes. Le problème survient quand nous souhaitons manipuler tous ces bâtiments en même temps, sans forcément connaître son type, par exemple pour l'afficher. A cause du typage, on ne va pas pouvoir créer un grosse liste pour l'ensemble des bâtiments et on devrait créer une liste pour chaque type de bâtiment, ce qui va vous forcer à dupliquer votre code et à le rendre moins "maintenable". Ici, nous allons utiliser une classe abstraite **`Building`** qui va venir englober tous nos bâtiments afin que l'on puisse les manipuler dans une seule et même structure. On ne manipulera plus les bâtiments par des listes distinctes : une liste de maisons, une liste de postes de police, etc. On aura une liste de bâtiment, sur laquelle on va pouvoir effectuer des opérations sans connaître le type de bâtiment qu'on manipule.

Voici comment se présente une classe abstraite:

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

On peut constater l'apparition du mot clé **`abstract`** qui vient compléter non seulement la définition de la classe, mais également les prototypes des méthodes propres à cette classe. Du fait que cette classe soit abstraite, elle ne peut pas être instanciée et toutes ses fonctions doivent être abstraites. On reviendra sur ces notions dans la partie consacrée à la construction de notre SimCity.

### L'héritage

L'héritage est une notion de la programmation orientée objet qui va venir compléter l'utilisation de nos classes abstraites. Effectivement, notre classe **Building** n'est pas instanciable et nous aimerions bien pouvoir créer des bâtiments. Nous allons donc créer plusieurs classes qui hériteront de la classe **Building**. Une classe **A** qui hérite d'une classe **B** hérite de tous les attributs et méthodes de **B**. Nous avons deux principaux avantages à l'utilisation de l'héritage: éviter la duplication de code et pouvoir manipuler des objets qui sont issus de classes différentes comme étant des objets du type de la classe dont ils héritent. Par exemple, si une classe **House** hérite d'une classe **Building**, alors nous pouvons manipuler un objet issu de la classe **House** comme étant un objet de la classe **Building**.

Voici un exemple de l'héritage:

```
1  class House : Building
2  {
3      private bool occupied;
4      private int inhabitants;
5      public House()
6      {
7          // Constructor ...
8      }
9      public override void Print()
10     {
11         // Code ...
12     }
13 }
```

Comme la classe **House** hérite de la classe **Building**, elle hérite également de tous ses attributs et ses méthodes. En l'occurrence, la classe **Building** possède l'attribut **cost** ainsi que la méthode **Print()**. On notera que lorsqu'une classe hérite d'une classe abstraite, alors vous avez l'obligation de fournir une implémentation pour toutes les méthodes de la classe abstraite. C'est ce qui va nous permettre d'avoir la garantie que tous les objets dérivés de notre classe abstraite implémentent leur propre version de ces méthodes et donc que l'on puisse les appeler. Ici, on doit donc redéfinir la méthode **Print()** de notre classe **Building** en ajoutant le mot clé **override** au prototype de la méthode.

### Les propriétés

On vous a appris à mettre les attributs de vos classes en privé et à les accéder par le biais de méthodes intermédiaires que l'on appelle *getters* et *setters*. Ces méthodes vous permettent de contrôler de votre côté comment vous allez servir vos utilisateurs et comment vous allez contrôler leur actions. Seulement, écrire deux méthodes par attribut privé est parfois un peu lourd, surtout quand vous allez commencer à avoir des classes contenant beaucoup d'attributs. De plus il est préférable d'accéder à un attribut

non par le nom d'une méthode (comme **GetAttribut()**), mais par le nom de l'attribut concerné seul. Cette semaine, nous allons donc voir une nouvelle méthode. Il s'agit (vous l'aurez deviné) des **propriétés**.

Elles se présentent sous cette forme:

```
1 class MyClass
2 {
3     private int myInt;
4     public int MyProperty
5     {
6         get { return this.myInt; } // Getter
7         set { this.myInt = value; } // Setter
8     }
9 }
```

Si on décompose la syntaxe, on obtient quelque chose du style :

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

Ce que l'on veut c'est que notre propriété soit visible en dehors de la définition de la classe, on mettra donc **public** devant la définition de notre propriété. Ensuite, de la même manière que pour une variable, on a un type et un nom que vous pouvez définir comme bon vous semble. Enfin, on terminera par écrire les accesseurs. On a deux mots clés **get** et **set** qui vont nous permettre de définir respectivement comment nous allons accéder à l'attribut et comment nous allons assigner une nouvelle valeur à l'attribut. Ce qu'il faut savoir c'est qu'en réalité, une propriété fait office d'interface entre un utilisateur (vous) et un attribut. Ce qui veut dire que l'on doit déclarer un attribut qui sera lui en privé, pour que l'on ne puisse pas y accéder depuis l'extérieur. Ici, on a déclaré l'attribut **myInt**. La propriété **MyProperty** servira donc à y accéder.

On peut utiliser cette propriété de la manière suivante:

```
1 public static void Main(string[] args)
2 {
3     MyClass monInstance = new MyClass(); // On instancie un nouvel objet
4     int x = monInstance.MyProperty; // x = 0
5     monInstance.MyProperty = 666; // myInt = 666
6 }
```

Bien entendu, on peut renommer **MyProperty** pour que le nom soit plus intuitif. Commentons un peu ce code. À la ligne 3, on créé un nouvel objet avec le constructeur par défaut, donc **myInt** vaut 0 pour cet objet. Ensuite on utilise notre propriété pour accéder en lecture à notre attribut et stocker sa valeur dans une variable à la ligne 4. Cet accès à l'attribut est défini dans la partie déclaré à côté du mot clé **get**. Puis ensuite, on utilise notre propriété pour accéder à notre attribut **myInt** mais

cette fois en écriture pour lui assigner une nouvelle valeur. L'assignation est définie quant à elle dans la partie à côté du mot **`set`**. Cette nouvelle valeur correspondra au mot clé **`value`** dans la définition de l'accesseur, ici 666.

Maintenant nous avons un autre problème. C'est toujours trop long de définir les accesseurs dans une propriété dans l'unique but de créer des accès normaux (sans traitement particulier) comme on vient de le faire. Heureusement, nous avons ce que l'on appelle des **propriétés auto-implémentées** qui vont grandement nous faciliter la vie.

La totalité de ce qui était défini dans **`MyClass`** peut se redéfinir comme ceci:

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

Oui c'est tout. Cette définition est équivalente à celle du dessus. *Attend une seconde, je vois pas où est l'attribut. Une propriété n'est pas censé être un moyen d'y accéder et non pas un champ en elle même ?* C'est exact, mais avec les propriétés auto-implémentées, un attribut est déclaré pour vous. Pas mal non ? Nous vous encourageons à utiliser les propriétés dans ce TP plutôt que des méthodes afin d'accéder à vos attributs.

N'hésitez pas à relire cette partie plusieurs fois, ou à poser des questions à vos ACDC si vous n'avez pas bien compris ces notions.

# 2 Les bases

Il est temps de se mettre au travail et de commencer à coder votre SimCity. Rassurez-vous, une fois fini, nous serons bien loin d'atteindre quelque chose qui ressemble à un vrai SimCity. C'est pour ça que la section Bonus sera large, ce qui vous permettra de nous impressionner et de vous exercer. Il est conseillé de lire tout le sujet avant de commencer. De cette manière, vous aurez une meilleure vue d'ensemble du projet. En regardant les précédents sujets, nous nous sommes dit que vous manquiez d'entraînement pour dessiner des maisons. Ne vous en faites pas, avec ce TP vous serez servis.

## 2.1 Des bâtiments

Pour commencer, nous allons faire une classe pour avoir des bâtiments. Cependant, à aucun moment nous ne créerons des objets représentant des *bâtiments*, mais à chaque fois des constructions plus précises, comme des maisons, des usines, etc. Vous devez donc implémenter une classe abstraite **`Building`**, qui va servir de point de départ pour toute les autres classes de bâtiment que nous allons créer. Nous vous conseillons (obligeons) de coder vos classes dans différents fichiers. Pour ajouter un nouveau fichier allez dans la vue d'ensemble de votre solution (Alt+Shift+S), clic-droit sur le dossier

de votre projet puis *Add* > *New File* > *General* > *Empty Class*. Nous ne voulons en aucun cas voir 5 classes dans un fichier. Soyez également sûrs de bien respecter l'architecture de rendu.

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

Dans le même fichier, déclarez une énumération **BuildingType** qui indiquera tous les types de bâtiments que le joueur pourra construire. Elle pourra avoir les valeur **House**, **CityHall**, **PoliceStation**, **Factory**, **Shop**. Vous pouvez ajouter des valeurs pour faire des bonus. Cette classe contiendra la méthode abstraite **Print()** qu'il faudra implémenter dans chaque classe fille. Elle doit également contenir un attribut pour stocker le type de bâtiment (cf. énumération).

## 2.2 Un dictateur

En tant que maire, vous vous devez d'avoir un QG digne de ce nom. Vous allez donc créer une classe **CityHall** qui va contenir les éléments principaux de votre mairie. **CityHall** hérite de la classe **Building** que nous avons construit tout à l'heure.

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

Vous n'oublierez pas d'implémenter les méthodes héritées de la classe **Building**, ainsi qu'un constructeur initialisant les variables de la classe.

Vous pouvez afficher votre mairie comme vous le souhaitez[2]. Cependant soyez raisonnables quant à la dimension de votre mairie afin de ne pas ruiner votre affichage final.

[2]Les ACDC qui ont écrit le sujet ne sont pas très bons en ASCII art.

Une mairie:

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

## 2.3 Des esclaves

Maintenant que vous avez votre palais, il vous faut un peuple. Créez une classe **House** qui contiendra un logement. Bien entendu, elle héritera de la classe **Building**.

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

Vous n'oublierez pas d'implémenter les méthodes abstraites héritées de la classe **Building**, ainsi qu'un constructeur initialisant les variables de la classe.

Une maison:

```
/----------\
|          |
| [ ]  [ ] |
|          |
|__________|
```

Chaque habitation pourra accueillir 30 habitants au maximum (vous n'avez pas assez de budget pour construire autre chose que des HLM). S'il n'y a plus assez de maisons, la population ne pourra pas s'agrandir.

## 2.4 Des gardiens de la paix

Le peuple peut être dissident Entre les manifestations, les grèves, les terroristes, la criminalité et l'opposition politique, il vous faut des hommes pour maintenir l'ordre et la discipline dans votre ville.

Implémentez une classe **PoliceStation** qui représentera, de manière évidente, un commissariat de police. Bien entendu, cette classe héritera de la classe **Building**.

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

Vous n'oublierez pas d'implémenter les méthodes héritées de la classe **Building**, ainsi qu'un constructeur initialisant les variables de la classe.

Un poste de police:

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

## 2.5 Au boulot tas de fainéants !

Votre peuple pourrait s'ennuyer. Mais heureusement, vous êtes là pour les aider.

Créez une classe **Factory** qui donnera du travail à tous vos habitants.

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

Vous n'oublierez pas d'implémenter les méthodes héritées de la classe **Building**, ainsi qu'un constructeur initialisant les variables de la classe.

Une usine:

Chaque usine pourra employer au maximum 100 habitants. S'il n'y a plus assez de travail, la population ne pourra pas augmenter.

## 2.6 La société de consommation

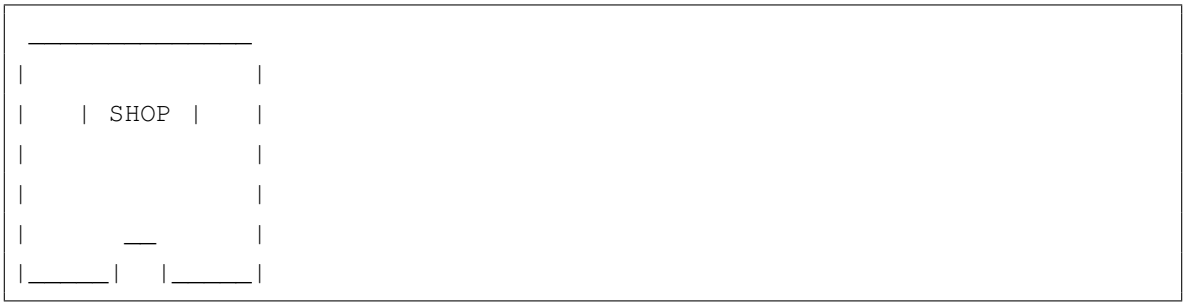
Vos habitants gagnent maintenant assez d'argent pour pouvoir le dépenser dans les commerces.

Créez une classe **Shop** qui hérite de **Building**.

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

Vous n'oublierez pas d'implémenter les méthodes héritées de la classe **Building**, ainsi qu'un constructeur initialisant les variables de la classe.

Un magasin:



Chaque commerce fournira 5 emplois à la ville, et pourra vendre à une population de 50 habitants maximum. S'il n'y a pas assez de commerces, la population ne pourra pas augmenter.

# 3 Construction de votre ville

Maintenant que vous avez les différents bâtiments, il vous faut construire votre ville.

Créez une classe **Town**.

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

Elle doit contenir trois méthodes : une pour construire un bâtiment, une pour en détruire un et une pour les afficher.

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

Les deux premières méthodes renvoient **true** si la construction/destruction a bien pu se faire. Vous devrez en plus implémentez les méthodes **PrintState()** qui affiche l'état de la ville sur la console (l'argent, le nombre de bâtiments et le nombre d'habitants), ainsi que la méthode **Update()** qui mettra à jour ces valeurs. La méthode **Update()** est le centre de votre jeu. C'est ici que vous allez coder tout le changement de votre ville en fonction de son état. Par exemple, l'argent va évoluer en fonction du nombre de magasin et du nombre de postes de police dans votre ville. Cette méthode sera appelée à maintes reprises dans la méthode **Main** afin de mettre à jour l'état de votre ville.

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

Exemple d'affichage de la méthode **PrintState** :

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

# 4 La méthode main

Maintenant nous allons compléter la méthode **main** afin de créer le jeu final. Pour vous aider, nous avons fait un récapitulatif des règles du jeu que vous devez implémenter:

- Vous démarrez le jeu avec une certaine somme d'argent et une ville sans bâtiment. Libre à vous de mettre en place vos propres valeurs de départ.

- Vous devez construire en tout premier votre mairie. Il ne peut y avoir qu'une seule mairie dans toute votre ville et vous ne pouvez pas créer de bâtiment sans avoir de Mairie.
- L'état de la ville devra être affiché par défaut sur la console.
- Le joueur pourra faire trois choix: construire un bâtiment, détruire un bâtiment, passer le temps (voir plus bas).
- La construction de bâtiment retire de l'argent au budget de la ville. Bien entendu, s'il n'y a pas assez d'argent, le bâtiment ne peut être acheté.
- La destruction d'un bâtiment vous rapporte une partie de son coût.
- Passer le temps est équivalent à afficher l'état de la ville au joueur sans lui afficher le menu des possibles actions. L'affichage est renouvelé à chaque changement d'état de la ville. Pour sortir de son état d'attente, il devra appuyer sur n'importe quelle touche.

Par rapport aux bâtiments, voici un récapitulatif de leurs comportements et des règles à implémenter:

- **`CityHall`**: le bâtiment principal de votre ville. Il est unique et ne peut pas être détruit. Il n'a pas de caractéristique spécifique.
- **`Factory`**: vous permet de créer 100 emplois pour votre ville. Si vous n'avez pas assez d'emploi à proposer, la population n'augmente plus.
- **`House`**: vous permet d'héberger 30 habitants au plus. La population ne pourra pas augmenter plus que ce que le nombre de maison ne le permet.
- **`PoliceStation`**: vous permet de limiter le taux de criminalité dans votre ville, ce taux de criminalité augmente en fonction de votre population. Nous dirons que les criminels représentent 30% de votre population (une vraie ville de pourris). A partir de 15% de taux de criminalité, la population de votre ville n'augmente plus. Un poste de police offre aussi 20 emplois à votre ville. Mais elle demande un certain entretien en équipement et va donc vous prendre un peu d'argent à chaque fois que la méthode **`Update()`** est appelée.
- **`Shop`**: vous permet de proposer à votre peuple des produits dont ils ont besoin pour survivre. Vous en profiterez également pour les taxer un maximum en imposant une TVA outrancière. Un magasin offre 5 emplois à votre ville et vous rapporte de l'argent à chaque appel de **`Update()`**.

Pour faire simple, votre méthode **`Main`** va ressembler à une grosse boucle où vous allez afficher votre menu pour que le joueur puisse sélectionner une action et afin d'appliquer les changements nécessaires à votre ville. Vous devez donc au préalable créer une ville afin de pouvoir appeler ses méthodes pour créer ou détruire ses bâtiments.

Il a déjà été dit précédemment que la méthode **`Update()`** de votre ville sera appelée plusieurs fois. Nous vous recommandons pour cela de mettre en place un timer[3] où vous appellerez votre méthode

[3] System.Timers

au bout d'un certain délai. Vous êtes libres de gérer le délai comme vous le souhaitez tant qu'il est inférieur à 10 secondes.

# 5 Bonus

Cette section est extrêmement libre. C'est-à-dire que vous pouvez faire absolument tout si vous estimez que ça améliore la qualité de votre jeu. Peut-être pouvez-vous le faire ressembler à un vrai SimCity. Tous les bonus doivent être expliqués dans le README. Voici quelques exemples de bonus que vous pouvez faire.

## 5.1 BRULEZ!!!

Nous vous rappelons que ce jeu est censé être un SimCity, et comme dans tout bon SimCity vous êtes un tyran et votre passe temps principal est de martyriser votre peuple. Du coup, vous allez créer des catastrophes naturelles afin de détruire des bâtiments aléatoirement et de tuer un bon nombre de citoyens. Après tout, ils n'avaient qu'à pas faire grève.

## 5.2 Affichage

Quand on joue à SimCity, on veut pouvoir placer nos bâtiments sur une carte. Vous allez donc faire un affichage avec une vue du dessus. Vous pouvez placer vos bâtiment à l'aide des flèches de la souris. Bien sûr, vous ne pourrez pas placer un bâtiment par dessus un déjà existant.

**The code is the law.**