# TP C#5 : ClassCity

```
login_x.zip
    | login_x/
        | AUTHORS
        | README
            | mySimCity.sln
            | mySimCity/
                    | Tout sauf bin/ and obj/
```

Rendu disponible : de Lundi 14/12/2015 09 : 00 à Dimanche 20/12/2015 23 : 42

Antoine Damhet <damhet_a@epita.fr>
Thomas Josso <josso_t@epita.fr>


# 1 Cours

## 1.1 The world of POO

L'objectif de ce TP est de découvir la programmation orientée objet (POO). Durant ce sujet vous allez découvrir des notions comme les *Classes* ou l'*Héritage*.

La POO est un paradigme : il s'agit d'une façon d'appréhender un problème algorithmique. Vous connaissez déjà l'approche fonctionnelle avec le Caml mais il en existe bien d'autre.

## 1.2 Classe & objets

Tout d'abord, regardons un exemple de déclaration de classe : *ACDC*.

- Attributs : les lignes 4, 5 et 8 sont des attributs. Ce sont des champs qui représentent les caractéristiques de l'objet que l'ont veut créer. Dans le cas présent on constate que les deux premiers ont le préfixe *public* alors que le dernier a le préfixe *private*. La différence est qu'un attribut *private* n'est accessible (en lecture ou écriture) que dans le namespace de sa classe.
- Getter & Setter : la ligne 12 est la déclaration d'une propriété *get* & *set*. Dans cet exemple le champs *nickname* n'est pas visible en dehors de sa propre classe. Avec la propriété *get* (public par défaut) de *Nickname* (N majuscule) on peut accéder à la valeur de la variable (et seulement par cette propriété !) hors de la classe. De la même manière *set* change l'écriture. Ici *private set* est redondant car *nickname* est déclaré privé, et n'est qu'un exemple.
- Constructeurr : la ligne 20 est la déclaration du constructeur. Cette fonction va créer un objet selon les caractéristiques de la classe. Le mot *this* sert à lever une indétermination lors d'une affectation d'un des champs de l'objet : *this* récupère le symbole "le plus proche". Attention ! Tous les champs de l'objet doivent être assignés lors de l'instanciation.
- Méthodes : la ligne 29 est la déclaration d'une méthode. Les méthodes sont des fonctions liés à tout objet du type de la classe dans laquelle elle est déclarée. Dans l'exemple ci-dessous *SayMyName* peut être éxécuté par tout objet de type *ACDC*.


```
1  public class ACDC
2  {
3      //Public Attributes
4      public string name;
5      public int year;
6
7      //Private Attributes
8      private string nickname;
9
10
11     //Getter & Setters
12     public string Nickname
13     {
14         get { return nickname; }
15         private set { nickname = value; }
16     }
17
18
19     //Constructor
20     public ACDC(string name, int promotion)
21     {
22         this.name = name;
23         year = promotion;
24         nickname = "default";
25     }
26
27
28     //Methods
29     public void SayMyName(void)
30     {
31         Console.WriteLine(this.nickname + ", you're god damn right.");
32     }
33 }
```


Les classes sont des patrons. Leur but est de générer des objets qui eux sont "vivants" : elles ne doivent être en aucun cas considérées comme des objets. Regardons un exemple concret.

```
1  //Giving life to an ACDC (the object)
2  ACDC acdc;
3  acdc = new ACDC("Thomas", 2018);
4
5  //If in the same namespace
6  acdc.nickname = "GiJo";
7
8  //Anywhere else where acdc is instantiated
9      //Call of an ACDC method
10     acdc.SayMyName();
11
12     //Use of an ACDC object property
13     Console.WriteLine(acdc.Nickname);
```

Ces notions sont fondamentales et sont la vraie force de la POO ! Vous pouvez créer des objets complexes (avec leurs propres méthodes) qui seront modifiés au cours du programme.


## 1.3 L'héritage

La notion d'héritage est un autre aspect de la POO en C#. C'est une façon de créer de nouvelles classes qui partagent les aspects de la classe mère PLUS leurs propres attributs et méthodes.
Imaginons que l'on veuille créer une nouvelle classe *ACDC2018* qui hérite de la classe *ACDC*. Dans cet exemple on considère la classe *ACDC* comme définie.

```
1 public class ACDC2018 : ACDC
2 {
3     public ACDC2018(string name)
4     {
5         this.name = name;
6         year = 2018;
7         nickname = "default";
8     }
9 }
```

Le nouveau constructeur de cette classe ne nécessite pas tous les paramètres précédents (aucun soucis là dessus). Pourtant ici Visual Studio va implicitement appeler le constructeur de la classe *ACDC* avant celui de la classe héritée *ACDC2018* et indiquer une erreur : *ACDC does not have a constructor that takes one argument.*

Ce que l'on veut plutôt faire :

```
1 public class ACDC2018 : ACDC
2 {
3     public ACDC2018(string name)
4         : base(name, 2018)
5     {
6         this.name = name;
7         nickname = "default";
8     }
9 }
```

De cette façon le nouveau constructeur peut être appelé avec des paramètres utilisateurs et par défaut !


Maintenant disons que nous n'utiliserons plus que la classe *ACDC2018*. La classe *ACDC* est devenue obsolète : nous ne créerons plus d'objets de ce type. *ACDC* n'est donc plus là que pour générer la classe qui hérite d'elle. Nous pouvons la rendre *abstract*.

```
1  public abstract class ACDC
2  {
3      public ACDC(string name, int promotion)
4      {
5          this.name = name;
6          year = promotion;
7          nickname = "default";
8      }
9  }
10
11 //Not possible anymore
12 ACDC acdc = new ACDC("GiJo", 2018);
13
14 //Valid
15 ACDC2018 best_acdc_eu = new ACDC2018("GiJo");
```

Un objet d'une classe abstraite ne peut plus être instancié. Bien évidemment les classes *abstract* n'ont d'intérêt que si plusieurs autres classes en héritent. Le code ci-dessus est un exemple.


## 1.4 Exceptions

Les exceptions sont un moyen d'assurer la stabilité de votre code. Elles peuvent être utilisées pour éviter des comportements non définis ou inattendus et des cas de sortie de programme. Une exception termine le programme en cours.

Si vous connaissez et voulez éviter certaines valeurs dans vos fonctions, vous pouvez lancer les exceptions correspondantes.

```
1  public static void SetYear(ACDC2018 acdc2018, int promotion)
2  {
3      if (promotion < 2000 || promotion > 2020)
4      {
5          throw new ArgumentException();
6      }
7      acdc2018.year = promotion;
8  }
```

Beaucoup d'autres exceptions existent (Google is your friend).

## 1.5 Try Catch

Un *Try-Catch* est une façon de tester du code dans la section *try* et d'éxécuter le code de *catch* en cas de sortie de programme.

```
1  public static void CodeTest(Student student2020)
2  {
3      try
4      {
5          student2020.Code();
6      }
7      catch (Exception e)
8      {
9          Console.WriteLine("The code has crashed !");
10         Console.WriteLine(e.Message);
11     }
12 }
```


# 2 Exercises

Dans ce TP nous allons créer une "ville". Oui une ville, avec ses batiments, sa population et même ses arbres (non pas des arbres binaires...).

## 2.1 Palier 0 : Créer le projet

Pour ce palier, vous allez créer le projet avec Visual Studio 2015. Attention, si vous ne passez pas ce niveau (autrement dit, si vous ne respectez pas l'architecture du rendu), vous ne serez tout simplement pas notés. Nous vous demandons de créer une Console Application dans Visual Studio 2015 Professional (Community et versions plus anciennes devraient marcher). File -> New -> Project -> Visual C# -> Console Application. Le projet et la solution doivent s'appeler "mySimCity".

## 2.2 Palier 1 : The city, the people and the houses

Dans ce niveau, vous allez créer 3 *classes* contenues dans :

— "City.cs"

— "Human.cs"

— "House.cs"

Chaque fichier devrait à présent resembler à ça :

```
1 // Des usings
2 namespace mySimCity
3 {
4     class MaClasse
5     {
6     }
7 }
```

Chaque *classe* devrait pouvoir s'initialiser, pour l'instant, ajoutez un constructeur vide ne prenant aucun paramètre (cela changera plus tard). Chaque classe devrait aussi pouvoir se décrire à l'utilisateur, chaque *classe* doit donc avoir une méthode *public void WhoAmI()* écrivant dans la console une courte description d'elle-même. Maintenant, nous voulons des gens dans notre belle ville, nous (je veux dire vous) allons ajouter 3 variables privées (avec le mot clé private) à la classe City : une liste de gens, de batiments


et un nom. Pour cela, vous allez utiliser la *classe List*, nous vous recommandons très fortement de consulter MSDN. Les listes peuvent être utilisées de cette façon :

```
List<int> maliste = new List<int>();
//On remplacera int par Human ou House dans notre cas
maliste.Add(0);
maliste.Add(1);
maliste.Remove(1);
//On peut afficher le contenu de la liste
foreach(int x in maliste)
{
        Console.WriteLine(x);
}
```

```
City ville = new City("Paris");
Console.WriteLine(ville.Name); // Doit afficher "Paris"
City.Name = "Lyons"; // Ne doit pas marcher
```

Maintenant nous devrions pouvoir ajouter et supprimer des éléments à nos listes. Vous devez créer 4 *méthodes* : *AddHuman(Human), AddHouse(House), KillHuman(Human), DestroyHouse(House).*

Un humain doit avoir quelques caractéristiques comme un nom et un âge (vous pouvez en ajouter d'autres). La *classe* Human doit être créée de cette façon *(Human(string, int)).*

```
Human h = new Human("Michel", 26);
```

Nous devrions être capable de faire cela :

```
int age = h.Age; // Age renvoit l'age de l'humain h
string name = h.Name; // Name renvoit le nom de h
h.BirthDay(); // C'est l'anniversaire de h, incrementer son age
h.Age = 10; // Ne doit pas etre valide
h.Name = "toto"; // Ne doit pas etre valide
```

House doit se comporter comme ceci :


```
House maison = new House(3);
int rooms = house.Rooms; // rooms = 3
house.Rooms = 6; // Ne doit pas marcher
```

Maintenant que nous avons rempli nos *classes*, nous (vous encore) pouvons créer une *méthode* qui décrit le contenu de l'*objet* : Describe().

```
City paris = new City("Paris);
paris.AddHuman(new Human("Michel", 26));
paris.AddHuman(new Human("Thomas", 21));
paris.AddHouse(new House(5));
paris.Describe();
```

Ce code doit afficher :

> Ville de paris
> 2 habitants :
> * Michel, 26 ans.
> * Thomas, 21 ans.
> 1 maison :
> * Maison de 5 pièces.

"habitants" et "maison" doivent être accordés. (Après le caractère '*', City doit appeler Human.Describe ou House.Describe).

## 2.3 Palier 2 : We want more

Nous (vous vraiment) allons ajouter les *classes* suivantes.

— Building. Elle hérite de House et à une propriété supplémentaire : Floors. Son *constructeur* doit être "Building(rooms, floors);", WhoAmI et Describe divent être modifiées en conséquences.

— Tree. Elle hérite aussi de House (yep) et doit toujours avoir 0 pièce. Son *constructeur* sera "Tree()" WhoAmI et Describe doivent être modifiées en conséquences.

— Male. Elle hérite de Human et doit juste avoir une version différente de WhoAmI et de Describe.

— Female. Comme pour Male.


```
City paris = new City("Paris);
paris.AddHuman(new Male("Michel", 26));
paris.AddHuman(new Female("Isabelle", 21));
paris.AddHouse(new Building(3, 5));
paris.AddHouse(new Tree());
House h = new House(2);
paris.AddHouse(h);
paris.DestroyHouse(h);
paris.Describe();
```

Ce code affiche :

> Ville de paris
> 2 habitants :
> * Michel, 26 ans (homme).
> * Isabelle, 21 ans (femme).
> 2 maison :
> * Immeuble de 5 étages de 3 pièces.
> * Je suis un aaaarrbree!!!

## 2.4 Bonus

Ce TP devrait être assez court pour que vous fassiez des bonus, surprenez nous. Vous pourriez ajouter un *objet* météore qui pourrait détruire une partie de la ville. Vous pouvez ajouter une carte de la ville qui contiendrait les coordonées des bâtiments, vous pouvez ajouter des rues ou tout ce à quoi vous pouvez penser.

And remember...

**Deadline is Coming**