TP C#7: My farm

1 Do you remember OOP ?

1.1 Object-oriented programming

In previous workshops you discovered what object-oriented programming is. Today we are going to learn heritage in C# . But before that here, are a few reminders:

Object-oriented programming (as known as OOP) is a programming paradigm. Programming paradigms are ways of thinking to solve problems. You already know one: functional programming since you learned Caml. There are lots of different programming paradigms (see https://en.wikipedia.org/wiki/Object-oriented_programming).

1.2 Lost objects

The main idea behind OOP is that everything can be represented as an **object**. Each object has its own features and can interacts with others. Before creating an object, we must declare it in a **class** : a class is a description of what the object is (attributes) and what can it do (methods). An object is an instantiation (a concrete version) of the class. You can use the following analogy to understand: a blueprint (the class) describes how a building should be. With it, we can build one or more buildings (objects). Look at the code below:

```
// Class definition
public Class ACDC
{
  // Public attributes
  public string gender;
  public float height;

  // Private attribute
  private int intelligence;

  // Class constructor
  public ACDC(float height, int intelligence)
  {
    gender = "unknown";
    this.height = height;
    this.intelligence = intelligence;
  }

  // Methods
  public void listen(string name)
  {
    Console.WriteLine("I listen you, " + name);
  }
```

```
    public void explain()
    {
      Console.WriteLine("RTFM");
    }

    public int attack(int nervousness)
    {
      return nervousness * 5;
    }

    public void run()
    {
    }
  }

  // Creation of an ACDC object
  ACDC acdc;
  acdc = new ACDC(0.41f, 1);
  // or
  ACDC acdc = new ACDC(0.41f, 1);

  // Acces and/or modification of an object field
  acdc.gender = "superhero";
  acdc.run();
```

You see, it is not difficult! Fortunately, OOP is considerably more powerful. Now, let's go for serious business.

2 Become a farmer

2.1 Dwarf and farm

Your old uncle O'Brien, who died recently, was a one-legged dwarf farmer who enjoyed dancing around totems. One evening, he received the prophecy indicating that you will inherit his farm at his death. So you are the proud owner of a small farm and you must take care of it, otherwise the curse will fall on you.

The problem is that you are not a good farmer, and it is not easy work. Fortunately your ACDCs are here to help you.

2.2 The project

The goal of this workshop is to build a standalone environment, your farm, where animals and plants will move according to specific rules. It is like a game of life, but the farmer version.

Work to do

Get the XNA's project on *http://perso.epita.fr/˜acdc*. You will find a Visual Studio solution in which the graphic part is already done. You must hand in the whole solution (without bin/ and obj/ folders) in a folder named as your ACDCs wish. Do not forget the AUTHORS file.

Of course, your code must compile, be indented and commented. Otherwise your grade will be null.

3 The farm animals

3.1 Your first animal

The first thing to do is to classify the different entities of the farm. It is important to know what we are talking about when we discover a new environment.

In a farm there are animals. These animals have some aspects in common. We can describe them in a class.

Create a new file called **Animal.cs** in which you will write a class `Animal` which has a private attribute `int nb_legs` and two public attributes: `pos_x` and `pos_y` of type `int`. You have to define a constructor:

```
public Animal(int nb_legs, int pos_x, int pos_y)
```

3.2 Animal inheritance

You have just created the `Animal` class but it would be nice to be a little more explicit. We will create the class `Pony`. This class will inherit from the `Animal` class.

Specifically, we will create a relationship between these two classes so that `Pony` inherits from `Animal`:

```
Class Pony : public Animal
{
  // Some code here
}
```

An inherited class means that the public fields that you defined in the `Animal` class will also be present in every `Pony` object ! It is as if all the code is set as public in the `Animal` class was copied to the `Pony` class (though this is not really the case).

What about private fields? Those are not inherited. For example, if you create a `Pony` object, it will not have a `nb_legs` field. Fortunately there is another visibility level between `public` and `private`: the `protected` level. It is used for fields which are only accessible from a class and its children (its heirs).

The most important thing to understand in inheritance is that it represents a relation **is a**. A pony is a particular animal, so the `Pony` class inherits from the `Animal` class. A motorbike and a car are specific vehicles. So we could create the `Motorbike` and `Car` classes that inherit from a `Vehicle` class. In poker, a color *is not* a card. So a `Color` class therefore does not inherit from a `Card` class.

Work to do

Change the code of `Animal` class so its child will be able to access to its private attributes. Create the `Pony` class in the file **Animal.cs**. This class inherits from `Animal`. Add a private attribute `awesome_lvl` to the class. For the moment this class does not compile. In the same file create the `Hen` class which also inherits from `Animal`. Add a protected attribute `nb_feathers`. This class also does not compile. What is the error ?

3.3 Default constructor

When an object of class `B` is instantiated, if `B` inherits from `A`, then the constructor of `A` is implicitly called.

```
public class B : A
{
  public B()
  // Implicit call of A constructor here
  {
    // Constructor of B
  }
}
```

The call of `A` constructor is done before instructions in the constructor `B()` are executed. If an object of type `Pony` is created, the constructor `Animal()` is implicitly called. The problem is that we redefined a constructor with the following prototype: `Animal(int nb_legs, int pos_x, int pos_y)`. This new constructor *hides* the default constructor `Animal()`. That is why Visual Studio says that `Animal` does not have a constructor which takes no argument.

3.4 Base constructor

The solution to this problem is to explicitly call a constructor of the base class with parameters. To call a method of the base class, we will use the keyword `base`. So we will call the constructor of `Animal` with three parameters. This code must be written before the braces because the `Animal` constructor is called before the execution of instructions in the `Pony` constructor.

```
public class A
{
  // Some code here
}

public class B : A
{
  public B(/* Some parameters here if you want */)
    : base.A (/* Some parameters here */)
  {
    //Some code here
  }
}
```

In this way we can call the wanted `A` constructor.

Work to do

Write the Pony constructor class by using this technique. You will call Animal constructor with 4 legs and pos_x and pos_y given as arguments to the Pony constructor. This constructor sets the private attribute awesome_lvl (the value must be between 0 and 41). Do the same thing for nb_legs but here nb_legs is 2. Choose a number of feathers between 301 and 2500.

```
public Pony(int pos_x, int pos_y)
public Hen(int pos_x, int pos_y)
```

4 Abstraction

4.1 Abstract class

If we create a class per animal, the Animal class has only one purpose: to provide a common base to all its subclasses, namely the attributes nb_legs, pos_x and pos_y. So instantiating the Animal class is not logical. We will make it non instatiable by making it *abstract.*

An abstract class is not instantiable. But its subclasses can be instantiated. The keyword to use is abstract.

```
public abstract class A
{
  // ...
}

A example = new A(); // Impossible
```

By doing this, A is *abstract.* You cannot create an object of type A. But you can create a new class which inherits from A.

```
public class B : A
{
  // ...
}

B example = new B(); // Possible
```

Work to do

Change your Pony class so it becomes an abstract class.

4.2 More inheritance, more fun

For the moment we have two classes, Pony and Hen that inherit from Animal. Since an inheritance relationship can have several levels, Animal can inherit from another class (abstract or not). And Hen can have subclasses.

Work to do

We provide you with the `Drawable` class that manages display. `Animal` have to inherit from `Drawable` because an animal **is a** drawable element. If you have problems with the `Drawable` class, make sure you are in the right `namespace myFarm`.

5 The ground

5.1 Ground inheritance

We will have two types of ground on which our animals will tread: dirt and grass. Dirt and grass have an **is a** relationship with the ground. This is much like what you have done for `Animal` and `Pony`/`Hen` so you know how it works.

Work to do

Create a new file **Ground.cs** and create an abstract class named `Ground`. Add two classes, `Dirt` and `Grass`, which inherit from `Ground`. These classes have an attribute `bool occupied` to know if the cell is occupied or not.

Think about what you have done. You have a `Ground` class, an empty abstract class, and two classes (`Dirt` and `Grass`) with the *same* attributes. In fact, `Ground` is useless! If you do things properly, you surely factorized common code of `Dirt` and `Grass` and have put it in `Ground`. Good ! But in this case `Dirt` and `Grass` are useless !

Here, two types of ground (dirt and grass) can simply be represented with only one non abstract class (`Ground`). This class will have an attribute called `type` which defines the type of ground.

5.2 Ground enumeration

For this attribute `type`, we will use an enumeration. Here is a reminder of enumeration syntax:

```
// Enumeration declaration
enum enum_ex
{
  enum_1,
  enum_2,
  enum_3
};

// Utilisation of the enumeration
enum_ex mon_enum = enum_1;
mon_enum = enum_2;
```

Work to do

In the file **Ground.cs** before the declaration of `Ground`, create the enumeration called `groundType` which has `dirt` and `grass` in the enumerator list. Change `Ground` to add the attribute `type`. The prototype of the constructor is:

```
public Ground(Vector2 pos, groundType ground_t)
```

6 Creation of the game

6.1 7up

It is time to take up the graphic rendering. This part has been started so you just have to complete it.
To draw an element, you will need some textures. These textures will be the same for each animal.
It is illogical to instantiate a class `Texture`, that is why it will be `static`.

Work to do

Go in the file named *Textures.cs* which contains a static class `Textures`. This class contains, as a static attribute, `pony_textures` of type `Texture2D`. Complete this class with the other animals, ground types and the farm.
`Texture` also contains a static method that allows you to load the correct png file for each texture. A folder of textures already exists for this project. These have been imported in the *myFarmContent (Content)* module, in a *Sprites* folder.
Follow the example to complete the `load()` method. In the file *Game1.cs*, call this static method `LoadContent()`.

```
Textures.load(Content);
```

The parameter given to `load()` is a content manager. It is an attribute of `Game`, of which `Game1` inherited.

6.2 Elements drawing

Now, we have to make our instantiable `Drawable` objects (farm, dirt, grass, etc...) with the correct texture. Let's begin with `Animal`.

Work to do

Modify the `Animal` constructor to make it take three parameters instead of two, the new one being a `drawable_type drawable_t` that is an enumeration containing all drawable elements. You should now call the `Drawable` constructor in the `Animal` constructor.

```
public Animal(uint nb_legs, Vector2 pos, drawable_type drawable_t)
    : base(drawable_t)
{
  // Some code here
}
```

Next, modify the `Pony` and `Hen` constructors to call the `Animal` constructor with the correct `drawable_type`. Do the same for `Ground`.

6.3 The farm

Create a new file *Farm.cs* containing a `Farm` class inheriting from `Drawable`. The farm has two attributes: `size_x` and `size_y` of type `int` and an array of type `Ground` called `grounds`.
As a reminder, the declaration of a two-dimensional array is:

```
// Declaration of a 2-dimensional array of type A
A[,] array;

// Initialisation of this array
array = new A[10,10];
```

The `Farm` constructor will take as parameters two integers, representing the width and height of the farm. The constructor will change the `size_x` and `size_y` attributes accordingly and instantiate the `Ground` array with the correct size. The ground has chance in three to be grass. You also have to call a method to add animals in the farm. This method will take a random cell and create an animal if it is free. The animal has a 50% chance to be a hen and 50% to be a pony.

Work to do

Implement the two following methods:

```
public Farm(int size_x, int size_y)
public void addAnimal()
```

Do not forgot to update the farm when putting an animal in a cell and to set it as occupied.

6.4 Draw me a farm

Now you have to draw all the elements. To display a `Drawable`, you only have to call its `Draw()` method.

Work to do

In the file *Farm.cs*, create the `display()` method that can print all the elements. Here is the prototype:

```
public void display(SpriteBatch sb)
```

In this method, you have to:

- Print the farm.
- Print the correct ground for each cell.
- Print the correct animal contained in the cell if there is one.

6.5 My farm

We are close to the end! Now, go in *Game1.cs* and add an attribute `my_farm` of type `Farm`. In `LoadContent()`, crate a new farm of size 10*10 and with 10 animals in it.
Finally, in the `Draw()` method, write:

```
// TODO: Add your drawing code here
spriteBatch.Begin();
my_farm.display(spriteBatch);
spriteBatch.End();
```

You are now a real farmer!

7 Next generation

7.1 Raising chicks

It is good to have hen but it will better to have chicks. To do so, create a new class `Chick` that inherits from `Hen`.

Work to do

Modify *Textures.cs* and *Drawable.cs* to handle chicks.

Create a chick

We need to create a constructor for the `Chick` class. If we write the following method...

```
public Chick(int feather, Vector2 pos)
```

we will have a little problem. In fact, the `Chick` constructor calls the `Hen` constructor that calls the `Animal` constructor with the *drawable_ type* of the `Hen`. So the `Chick` will also have the texture of a hen...
A possible solution is to overload the `Hen` constructor with a method that takes as third parameter a *drawable_ type* as follows:

```
public Hen(int feathers, Vector2 pos, drawable_type drawable_t)
```

Work to do

Go in the *Animal.cs* file and implement the new constructor of `Hen`. Then, create the `Chick` constructor and modify the `addAnimal()` method to generate some chicks.

8 Animation in the farm

8.1 Animals update

A static farm is not interesting, so let's make it more attractive.
Here are some rules to make your animals move:

- A pony moves from `(awesome_lvl % 2) + 1)` cells in one of the eight directions from its cell.
- A hen moves from one cell (or two cells if the number of feathers is a multiple of three) in one of the eight directions from its cell.
- A chicken moves from one cell in one of the eight directions if its number of feathers is a multiple of two. Every turn, its number of feathers can increase of eleven. If its number of feathers exceeds 300, it becomes a hen.
- Do not forgot to update the state of the cell and to check if the new cell is out of the range of the array.

Work to do

Go in class `Animal` and add the `Update()` method. Let us assume that an animal does nothing when it is updated. So, if we do not implement an `Update()` method in a class that inherits from `Animal`, instantiated objects will have this behavior.
For our specific animals, a custom `Update` method will be called. The keywords *virtual* and *override* were made for this. They will hide the base class method to use the child's one. That is polymorphism.

```
class A
{
  public virtual void method(/* Some parameters */)
  {
    // Some instructions
  }
}

class B : A
{
}

class C : A
{
  public override void method(/* The SAME parameters */)
  {
    // Some others instructions
  }
}
```

```
// Some examples
A ex_1 = new A();
B ex_2 = new B();
C ex_3 = new C();
C ex_4 = new A();

ex_1.method(/*...*/);    // call A's method
ex_2.method(/*...*/);    // call A's method
ex_3.method(/*...*/);    // call C's method
ex_4.method(/*...*/);    // call C's method
```

There is another keyword related to polymorphism: `new`. Contrary to `override`, it completely hides the base class. `new` still allows the access to an inherited class if the object is interpreted as a base class.

```
class A
{
  public void method(/* Some parameters */)
  {
    // Some instructions
  }
}

class B : A
{
  public new void method(/* The SAME parameters */)
  {
    // Some other instructions
  }
}

// Some examples
B ex_1 = new B();
A ex_2 = new B();

ex_1.method(/*...*/);     // call B's method
ex_2.method(/*...*/);     // call A's method
(A)ex_2.method(/*...*/);  // call A's method
```

You need to be aware that *overloading* is not *polymorphism*. Overloading consists in two different methods that have the same name but a different signature. Polymorphism consists in a base class on that has a certain method and a son class with a method with the same signature. In our project, we have an array of type Animal and we want to call the specific method to each class. To do so, we need to use *virtual* and *override*.

Work to do

Implement the necessary `update(Farm farm)` methods.

```
// In Animal class
public virtual void update(Farm farm)

// In inherited classes
public override void update(Farm farm)
```

The order of actions to be executed:

```
// Update the Animal
// Previous ground is now not occupied
// New ground is now occupied
// New ground contains the animal we are talking about:
farm.grounds[new_x, new_y].containing = this;
```

8.2 Update of the farm

Let us create an `update()` method in the Farm class.

```
public void update();
```

Do not update an animal that has just moved because it will be updated twice in the same turn. The best way to do it is to add an *nb_updates()* attribute for the farm and each animal. When the farm updates, it increments this attribute. Before updating an animal, we compare if the number of updates of the farm is heigher than the number of updates of the animal. If so, increment the `nb_updates` attribute of the animal and update it.

8.3 Move that chick

A chick has to have more than 300 feathers to become a hen. When the number of feathers is sufficient, tranform the chick into a hen in the new cell.

Work to do

Create a new constructor for the Hen class that take a Chick as parameter.

```
public Hen(Chick previous)
```

This constructor will give the same number of feathers and and the same number of updates to the new hen. Then, create a new hen for the farm:

```
farm.grounds[new_x, new_y].containing = new Hen(this);
```

8.4 A new state

Finally, update the farm when the touch *Enter* is pressed. To do so, go in *Game1.cs* and print this in the `update()` method.

```
if (Keyboard.GetState().IsKeyDown(Keys.Enter))
    my_farm.update();
```

Congratulations, you now have a farm with ponies, hens and chicks!

9 Bonus

9.1 More and more animals

Here are some examples of animals that you can add to the farm:

- Cows have an integer attribute `nb_spots` somwhere between 5 and 10. They head to the grass from one cell per turn and it becomes dirt. If a cow is surrounded by dirt, it moves randomly from two cells in one of the eight direction.
- Pigs stay on dirt and avoid grass. If they are surrounded by grass, they cannot move, otherwise, they move from one cell.
- Roosters head to the nearest hen. When they are on the same cell, the hen heads to the top cell and the rooster the bottom one and a chick appears between them.
- Now that the rooster exists, the chick can evolve in Rooster. It has a 60% chance to become a hen and 40% a rooster.
- You can also handle life and death. When an animal exceeds 100 turns, it dies and has a 70% of chance of creating a new animal in the current cell.

Of course, you can create your own animals!

9.2 Plants

A farmer does not just have animals, it can also grow crops!
Handling crops implies that every cell can now be updated.

- At each update, a dirt cell has a 25% chance to become grass.
- You can plant corn. Every grass cell has a 10% chance to become a sprout of corn. It makes 3 turns to become corn. Then, every pig heads to the corn cell and the first one eats it. The cell becomes dirt.
- If the whole farm is reduced to dirt, all animals die and it is the end of the game.