My herited farm

"What is the object oriented way of getting rich ?"

1 Classe préparatoire

1.1 La programmation orientée objet

Lors des précédents TPs, vous avez pu avoir des aperçus de ce qu'était la programmation orientée objet. Nous allons aujourd'hui nous y intéresser de manière plus détaillée. Mais avant cela quelques rappels ne feront pas de mal.

La Programmation Orientée Objet (ou POO) est ce qu'on appelle un paradigme de programmation. Les paradigmes de programmation sont une manière de penser le code pour résoudre les solutions aux problèmes posés. Vous avez déjà vu un paradigme puissant, la programmation fonctionnelle avec OCaml. Vous travaillez en TD avec le langage algo qui est un langage procédural. Vous allez maintenant commencer à découvrir le monde de l'objet.

1.2 Objets perdus

Le principe de la POO est que tout peut être représenté comme un **objet**. Chaque objet est une entité qui possède ses propres spécificités, et qui peut interagir avec les autres objets au moyen d'actions. Pour créer un objet, on doit le définir auparavant dans une **classe** : une classe est la *description* d'un objet où l'on définit ce que c'est (attributs) et ce que ça peut faire (méthodes). Un objet est une instanciation d'une classe (une version concrète). On peut utiliser l'analogie suivante : un plan (la classe) décrit comment doit être un bâtiment. Avec ce plan on peut construite un ou plusieurs bâtiments (les objets). Regardez un peu le code ci-dessous :

```
// 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();
```

Vous voyez, ce n'est pas si compliqué ! Mais heureusement, ce n'est pas tout. La POO est bien plus riche que cela, c'est maintenant qu'on va s'amuser.

2 Farmer c'est la vie

2.1 Une histoire de nain et de ferme

Ce matin, alors que vous vous réveilliez tranquillement pour aller à votre TP de C# vous apprenez la disparition de votre oncle éloigné, le vieux O'Brien. C'était un fermier nain unijambiste qui s'amusait à danser autour de ses totems tous les soirs de pleine lune enroulé dans du jambon. Un de ces soirs, il reçut la prophétie indiquant de vous léguer sa ferme à sa mort. Vous vous retrouvez donc avec une petite ferme sur les bras, dont vous devez vous occuper sous peine de malédiction.

Mais voilà, vous n'êtes pas très doué en culture fermière, c'est que c'est pas évident ! Heureusement, vous étiez sur le point d'aller voir vos ACDCs.

2.2 Le projet du jour

Le but de ce TP est de créer un environnement autonome, votre ferme, où des animaux et autres plantes bougeront selon des règles spécifiques. On peut dire que c'est un jeu de la vie, farmer edition.

Travail à faire

Récupérez le projet xna à l'adresse suivante : *http ://perso.epita.fr/~acdc*. Vous y trouverez une solution où une partie graphique a été faite. Bien sûr on vous a réservé le meilleur. Vous devrez rendre toute la solution (en dehors des dossiers bin et obj) dans un dossier dont vos ACDC choisiront le nom, avec bien sûr le traditionnel AUTHORS. Pas besoin de préciser que votre code doit compiler, être indenté et et bien commenté sous peine de perdre des points sur votre note.

3 Les animaux de la ferme

3.1 Un animal de première classe

La première chose à faire est de classer les différentes entités qui sont dans la ferme. Savoir de quoi on parle est primordial lorsqu'on découvre un nouvel environnement !
Dans une ferme il y a tout d'abord des animaux. Ces animaux ont certains aspects en commun, que nous pouvons décrire dans une classe.

Créez un nouveau fichier appelé **Animal.cs** et implémentez une classe `Animal` qui possède comme attribut privé `int nb_legs`, et comme attribut public `pos_x` et `pos_y` de type int. Vous devez définir le constructeur qui aura la forme

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

3.2 Héritage d'animal

Vous venez de créer une classe animal, mais ça serait bien si on était un peu plus explicite. Nous allons créer une classe `Pony` qui va *hériter* de la classe `Animal`.

Concrètement qu'est-ce que ça donne ? On va créer une relation entre la classe Animal et la classe Pony, stipulant que `Pony` *hérite* de `Animal`.

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

Une classe héritée d'une autre signifie que les champs publics que vous aviez définis dans la classe `Animal` seront également présents dans tout objet de type `Pony` ! C'est un peu comme si tout le code mis en public dans la classe Animal était copié dans la classe Pony (mais heureusement ce n'est pas vraiment le cas).

Et les champs privés ? Ceux-là ne sont pas hérités. Par exemple, si vous créez un objet `Pony`, il n'aura pas d'attribut `nb_legs`. Heureusement, il existe un autre niveau de visibilité entre `public` et `private` : c'est le niveau `protected`. Il sert justement à ce que les champs d'une classe ne soient accessibles qu'à partir de la classe elle-même ou de ses filles (les classes héritières).

Le plus important dans la compréhension de l'héritage est qu'elle permet une relation entre deux classes de type **est-un** : un poney *est un* animal particulier, il hérite donc de celui-ci. De même une moto et une voitures sont des véhicules particuliers. On pourrait donc créer une classe `Moto` et une classe `Voiture` qui hériterait d'une classe `Véhicule`. Au contraire, au poker, une couleur *n'est pas* une carte. Une classe `Couleur` n'hérite donc pas d'une classe `Carte`.

Travail à faire

Modifiez le code de la classe `Animal` pour qu'elle puisse faire hériter ses attributs privés. Créez la classe `Pony` dans le fichier **Animal.cs** de sorte qu'elle hérite de la classe `Animal`. Donnez-lui un attribut privé `awesome_lvl`. Pour l'instant la classe ne compile pas. Créez ensuite la classe `Hen` dans le même fichier, qui hérite également de la classe `Animal`. Donnez-lui un attribut protected `nb_feathers`. De même, elle ne compile pas non plus. Quelle est l'erreur de compilation ?

3.3 Constructeur par défaut

Lors de l'instanciation d'un objet de la classe B qui hérite de la classe A, le constructeur de A est implicitement appelé.

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

L'appel au constructeur de `A` se fait même *avant* les instructions pour construire un objet de type `B`. Si un objet de type `Pony` est créé, le constructeur `Animal()` est implicitement appelé. Hors nous avons redéfini un constructeur dont le prototype est `Animal(int nb_legs, int pos_x, int pos_y)`. Ce nouveau constructeur *cache* le constructeur par défaut `Animal()`, c'est pourquoi Visual Studio nous dit qu'`Animal` ne contient pas de constructeur qui accepte 0 argument.

3.4 Appel du Constructeur de Classe, c'est la classe

Une solution consiste à préciser lors de la construction de la classe fille que le constructeur de la classe mère attend des paramètres. Pour cela, il faut écrire qu'on veut appeler une méthode définie dans la classe mère. Le mot-clé à utiliser est `base`. On va donc expliciter que l'on veut appeler le constructeur de `Animal` avec trois paramètres. Ce code n'est pas à écrire entre les accolades : ce qui est entre accolades se fait une fois que l'objet a été instancié. Il faut donc l'écrire avant cela :

```
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
  }
}
```

De cette manière on appelle le constructeur de A que l'on veut.

Travail à faire

Créez le constructeur de la classe `Pony` en utilisant la méthode ci-dessus. Vous appellerez le constructeur de `Animal` avec comme paramètres 4 pattes, ainsi que `pos_x` et `pos_y` donnés en argument au constructeur de `Pony`. Le constructeur remplit aussi l'attribut privé `awesome_lvl` de `Pony`, lui donnant une valeur entre 0 et 41. La même chose est à faire pour `Hen`, qui appellera le constructeur de `Animal` avec un `nb_legs` égal à 2. Vous lui attribuerez un nombre de plumes compris entre 301 et 2500.

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

4 Mise en forme abstraite

4.1 Abstraction de Classe

Quand on y réfléchit, si on crée une classe pour chaque animal, la classe `Animal` en elle-même ne sert qu'à ce que ses classes héritières aient un socle commun, à savoir les attributs `nb_legs`, `pos_x` et `pos_y`. Par contre, instancier la classe `Animal` serait peu logique. On va donc la rendre non instanciable en la rendant *abstraite*.

Une classe abstraite ne peut être instanciée. Par contre, ses classes filles sont instanciables. Le mot clé à utiliser est `abstract`.

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

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

En faisant ceci, vous rendez votre classe `A` *abstraite*. Elle ne peut plus être instanciée. Dans l'exemple ci-dessus, si vous faites hériter une classe B de A, celle-ci n'est pas abstraite.

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

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

Travail à faire

Modifiez votre code pour rendre la classe Animal abstraite.

4.2 Abstraction d'abstraction de Classe et héritage d'héritage de Classe

Pour l'instant nous avons vu deux classes, `Pony` et `Hen`, qui héritent de `Animal`. Les relations d'héritages pouvant comporter plusieurs niveaux, il est aussi possible pour `Animal` d'être elle-même héritée d'une autre classe, abstraite ou non. De même, la classe `Hen` peut elle-même avoir des héritières.

Travail à faire

Nous avons déjà codé une classe `Drawable` qui gérera l'affichage pour vous. Faites dériver la classe `Animal` de `Drawable`. Un animal **est-un** élément dessinable, on peut donc créer ce lien d'héritage. Si votre programme ne reconnait pas la classe `Drawable`, vérifiez si vous avez tout mis sous le même `namespace myFarm`.

5 Le sol

5.1 Héritage de terrain

Pour le sol sur lequel nos animaux vont se déplacer, nous allons créer deux types de terrain : de la terre et de l'herbe. Les deux ont une relation **est-un** avec le sol. Au final cela ressemble assez à ce que vous avez fait avec `Animal` et `Pony`/`Hen`, vous savez ce qu'il faut faire !

Travail à faire

Créez un nouveau fichier **Ground.cs** et créez la classe abstraite `Ground`. Faites-lui hériter deux classes, `Dirt` et `Grass` qui contiennent chacun un attribut `occupied` de type `bool` qui symbolise si la case est occupée, et un attribut `containing` de type `Animal`, qui contiendra un animal si la case est occupée.

Maintenant réfléchissez à ce que vous venez de faire. Vous avez une classe `Ground` abstraite qui ne contient rien, et deux classes `Dirt` et `Grass` qui héritent de `Ground` et qui contiennent *exactement* les même attributs. En fait, la classe `Ground` ne sert à rien ! Si vous avez fait les choses proprement, vous avez sûrement factorisé le code commun aux classes `Dirst` et `Grass` et l'avez mis dans la classe `Ground`. C'est bien mais dans ce cas ce sont `Dirt` et `Grass` qui ne servent à rien !

Le but de la Programmation Orientée Objet est de symboliser les choses au travers d'objets. Seulement, il ne faut pas forcément tout symboliser comme étant un objet. Réfléchissez à deux fois avant de créer une nouvelle classe et demandez-vous s'il est vraiment nécessaire d'avoir un niveau d'héritage de plus.

Ici, les deux types de terrain (terre et herbe) peuvent simplement être modélisées avec juste une classe `Ground` qui ne sera pas abstraite, et qui contiendra un attribut public qui porte comme nom `type` et qui permettrait de définit s'il s'agit de terre ou d'herbe.

5.2 Énumération de sol

Pour l'attribut `type` nous allons utiliser une énumération. Petit rappel sur la syntaxe :

```
// 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;
```

Travail à faire

Dans le fichier **Ground.cs** au dessus de la classe `Ground`, créez une énumération qui porte le nom `groundType` et qui possède comme possibilités `dirt` et `grass`. Modifiez la classe `Ground` pour lui ajouter l'attribut `type` de type `groundType`. Le prototype du constructeur est le suivant :

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

6 Création du jeu

6.1 7up

Bien, il est temps de s'occuper du rendu graphique. Cette partie a déjà été grandement effectuée. Vous n'aurez qu'à remplir quelques parties.
Pour afficher un élément, nous avons besoin notamment de textures 2D. Ces textures seront les même pour chaque animal : nous auront une texture pour 50 poneys, et non une texture par poney.
Il est illogique de pouvoir instancier une classe Textures alors que celle-ci ne sert qu'à contenir des sprites qui n'ont pas besoin d'être instanciés. C'est pourquoi celle-ci est **static**.

Travail à faire

Allez dans le fichier appelé *Textures.cs*, qui contient une classe statique Textures. Cette classe contient comme attribut statique *pony_ textures* de type `Texture2D`. Complétez la classe avec les autres animaux, les types de sol et la ferme.
Textures contient aussi une méthode `static` qui permet de charger les bons fichiers png pour

chaque texture. Un dossier de textures existe déjà pour le projet, ceux-ci ont été importés dans le module *myFarmContent (Content)*, dans un dossier *Sprites*.
Suivez l'exemple pour compléter la méthode load(). Dans le fichier *Game1.cs*, appelez cette méthode statique `LoadContent()` :

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

Le paramètre passé à `load()` est un Content manager, qui sert à charger une image depuis le contenu de notre projet. Ce paramètre est un attribut de `Game`, dont `Game1` hérite.

6.2 Dessin d'éléments

Maintenant, il va falloir faire en sorte que nos objets Drawable (farm, dirt, grass etc) soient instanciables avec la bonne texture. On va commencer avec Animal.

Travail à faire

Modifiez votre constructeur de `Animal` pour qu'il prenne maintenant 3 paramètres au lieu de 2, le nouvel étant un `drawable_type drawable_t`, qui est une énumération contenant tous les types d'éléments dessinable. On a par exemple `drawable_type.pony` ou `drawable_type.hen`. Il sert pour la classe `Drawable`, vous devrez donc appeler le constructeur de `Drawable` dans le constructeur de Animal :

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

Modifiez ensuite le constructeur de `Pony` et celui de `Hen`, pour qu'ils appellent Animal avec le bon drawable_type. Faites de même pour la classe Ground.

6.3 La ferme !

On a les sols, on a les animaux, il ne reste plus que la ferme ! Créez un nouveau fichier appelé Farm.cs contenant une classe Farm qui hérite de Drawable (à nouveau vérifiez si vous êtes dans le bon namespace si vous ne voyez pas Drawable dans la liste déroulante). La ferme contient deux attributs *size_x* et *size_y* de type int, ainsi qu'un tableau de Ground appelé *grounds*. Pour rappel, la déclaration d'un tableau à deux dimensions se fait ainsi :

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

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

Le constructeur de Farm prendra en paramètre deux int représentant la taille de la ferme. Le constructeur va affecter les attributs *size_x* et *size_y*, va créer un tableau de Ground de la bonne taille, et va l'initialiser avec des dirts et des grass. Pour itérer dans le tableau, utilisez deux boucles imbriquées ! Vous avez 1 chance sur 3 pour que le sol soit de l'herbe.
N'oubliez pas qu'il faut aussi appeler le constructeur de Drawable avec *drawable_t*.

Il faudra également une méthode qui permet d'ajouter un animal dans la ferme. Celle-ci prend une case au hasard dans la ferme et vérifie qu'elle est libre. Si elle l'est, elle crée un nouvel animal : 50% de chance pour que ça soit un poney, et 50% pour que ça soit une poule.

Travail à faire

Implémentez les deux méthodes décrites ci-dessus. Les prototypes sont les suivants :

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

N'oubliez pas de bien mettre à jour la ferme en mettant un animal dans la case, et en rendant cette case occupée.

6.4 Dessine-moi une ferme

Il ne reste plus qu'à dessiner les éléments. Pour dessiner un Drawable, il suffit d'appeler la méthode Draw() de ce drawable.

Travail à faire

Dans le fichier Farm.cs, créez la méthode display() permet de tout dessiner d'un coup. Voici le prototype :

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

Elle fait dans l'ordre :
-Affichage de la ferme,
-Pour chaque case du tableau, affiche le sol,
-Si la case est occupée, affiche l'animal contenu dans la case.

6.5 Ma ferme à moi

On y arrive ! Allez dans le fichier Game1.cs. Ajoutez un attribut my_farm de type Farm. Dans LoadContent(), créez une nouvelle ferme en utilisant le mot-clé new. Les attributs à passer sont 10 et 10 pour avoir un joli rendu. Ensuite, ajoutez 10 animaux dans la ferme.
Enfin, dans la méthode Draw(), écrivez ceci :

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

Vous voici l'heureux héritier d'une ferme.

7 Next generation

7.1 Élevage de poussins

C'est bien d'avoir des poules, mais ça serait bien aussi si on pensait à l'avenir. Nous allons créer une nouvelle classe Chick qui hérite de Hen. Vérifions la légitimité de l'héritage : Un poussin **est-une** poule (ou presque, pour des raisons de modélisation on dira que c'est close enough).

Travail à faire

Modifiez les fichiers Textures.cs et Drawable.cs pour gérer les poussins.

Construire un poussin

Qu'a un poussin de plus qu'une poule ? Pas grand chose. Au niveau des attributs, ils ne changent pas. Par contre, on peut penser qu'au niveau des méthodes, elles diffèrent légèrement. Il va quand même falloir un constructeur pour Chick. Si on écrit la méthode suivante :

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

On aura un petit problème : Le constructeur de Chick appellerait le constructeur de Hen, qui appellerait le constructeur d'Animal avec le drawable_type de la poule, ce qui fait qu'au final le poussin aurait aussi la texture de la poule.
Une solution possible est de **surcharger** le constructeur de poule avec une méthode qui prend en troisième paramètre un drawable_type. La surcharge de méthode, ou l'**overloading**, signifie qu'une méthode peut avoir plusieurs implémentations si chaque signature (= nombre et type des paramètres) est différente.
Ainsi, on appellerait le constructeur de Hen avec trois paramètres lorsqu'on instancie Chick, et le prototype du constructeur de Hen serait le suivant :

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

Travail à faire

Allez dans le fichier Animal.cs et implémentez le nouveau constructeur de Hen. Faites dériver une classe Chick de Hen. Créez ensuite le constructeur de Chick. Nous n'avons pas non plus modifié la méthode *addAnimal()* pour qu'elle génère des poussins. Modifiez-là.

8 Animation dans la ferme !

8.1 Mise à jour des animaux

Histoire d'ajouter un peu d'interactivité, il serait intéressant de pouvoir animer la ferme. Voici quelques règles pour faire bouger vos animaux :
-Un poney se déplace de ((awesome_lvl % 2) + 1) cases dans une des huit directions à sa portée.
-Une poule se déplace d'une seule case dans une des huit directions, et de deux si le nombre de ses plumes est multiple de 3.
-Un poussin ne bouge d'une case dans une des huit directions que si le nombre de ses plumes est pair. À chaque tour son nombre de plume a une chance sur deux d'augmenter de 11. S'il dépasse 300 plumes il devient une poule.
-Il ne faudra pas oublier de tester si la nouvelle case est en dehors des limites ou si elle est déjà occupée, et de modifier les valeurs *occupied* des cases correspondantes.

Travail à faire

Allez dans la classe Animal et ajoutez une méthode publique Update(). Étant donné qu'on a besoin de savoir si les cases adjacentes sont occupées et ensuite de mettre à jour la position de l'animal, il va falloir que la méthode ait accès à la ferme.

Que faut-il écrire à l'intérieur ? La définition de la méthode pour les poneys ? Celle pour les poules, pour les poussins ? Les trois ? Partons du principe qu'un animal ne fait rien quand il se met à jour. Ainsi, si on n'implémente pas de fonction *update()* dans une classe qui hérite d'Animal, les objets instanciés auront ce comportement de base.
Pour nos animaux spécifiés, il va falloir que la méthode *update()* personnelle soit appliquée. Les mots-clé **virtual** et **override** sont là pour ça : ils vont cacher la méthode de la classe mère pour imposer celle de la classe fille. On parle alors de **polymorphisme**.

```
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
```

Il existe un autre mot-clé en rapport avec le polymorphisme : **new**. Contrairement à *override* qui cache complètement la méthode de la classe mère, *new* permet d'avoir encore accès à celle-ci si l'objet est interprété comme un type mère.

```
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
```

Il faut faire attention de ne pas confondre *surcharge* et *polymorphisme* : la surcharge consiste à avoir plusieurs méthodes portant le même nom mais ayant une signature différente. Appeler l'une ou l'autre dépendra des paramètres qui seront passés. Le polymorphisme consiste à avoir une classe mère avec une certaine méthode ayant une signature et une classe fille avec cette méthode **possédant la même signature**. Appeler l'une ou l'autre dépendra de la façon dont on interprétera l'objet.
Dans notre projet, nous avons un tableau d'Animals et nous voulons appeler les méthodes spécifiques à chaque classe. il faudra donc utiliser *virtual* et *override*.

Travail à faire

Écrivez les méthodes update(Farm farm) nécessaire.

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

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

L'ordre des actions à faire est :

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

8.2 Mise à jour de la ferme

Nous allons également créer une méthode *update()* dans la classe ferme, pour que la seule chose à appeler de l'extérieur soit my_farm.update().

Travail à faire

Implémentez la fonction *update()* qui met à jour chaque case qui contient un animal. Le prototype est :

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

Attention à ne pas mettre à jour un animal qui vient de se déplacer d'une case à côté, et qui serait donc mis à jour plusieurs fois le même tour. Le mieux serait d'avoir un attribut *nb_updates* pour la ferme et pour chaque animal. Quand la ferme se met à jour elle incrémente cet attribut. Avant de mettre à jour un animal, on compare si le nombre d'updates de la ferme est supérieure à celui de l'animal. Si c'est le cas, on incrément le nb_update de l'animal et on le met à jour.

8.3 Évolution de poussin

Pour que le poussin évolue, il faudra vérifier le nombre de ses plumes. S'il dépasse 300, au lieu de mettre un nouveau poussin dans la nouvelle case, il faudra mettre une poule. Seulement, Comment faire en sorte que les attributs du poussin soient transférés à la poule ?

Travail à faire

Créez un constructeur pour Hen, qui prend un poussin en paramètre.

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

Ce constructeur donnera à la nouvelle poule le même nombre de plumes et le même nombre d'updates.
Ensuite, il suffira de créer une nouvelle poule pour la ferme :

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

8.4 Entrée dans un nouvel état

Enfin, mettons à jour notre ferme lorsque nous appuyons sur la touche Entrée. Allez dans Game1.cs et tapez ceci dans la méthode *update()* :

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

Félicitations, vous avez maintenant une ferme avec des poney, des poules et des poussins qui bougent.

9 Bonus

Vous vous sentez l'âme d'un fermier ? Voici quelques challenges en plus à considérer.

9.1 More and more animals !

Avoir plus d'animaux serait quand même plus intéressant. Voici quelques exemples d'animaux à importer dans votre ferme et ce qu'elles font pour bouger :

- Les vaches ont un attribut nb_spots compris entre 5 et 10. Elles se dirigent vers l'herbe d'une case à la fois pour la brouter. Celle-ci devient de la terre. Si la vache est entourée de terre, elle bouge aléatoirement de deux cases dans une des 8 directions.

- Les cochons restent toujours sur de la terre et évitent l'herbe le plus possible. Si ils sont entourés d'herbe, ils ne bougent pas. Sinon, ils bougent d'une case.

- Les coqs se déplacent d'une case vers la poule la plus proche d'eux, peu importe la distance. Une fois sur la même case, la poule se dirige vers le haut et le coq vers le bas, et sur la case sur trouve un poussin. Si la poule ou le coq ne peut aller là ou elle/il doit aller, elle/il cherche une autre case adjacente.

- Maintenant que le Coq existe, il faudrait que notre poussin puisse évoluer dans les deux formes. Ce qui signifie qu'il nous faut un nouveau niveau d'abstraction : la classe Poultry, dont héritera les trois classes de volaille. Le poussin évolue à 60% en poule et à 40% en coq.

- On pourrait aussi gérer de manière plus réaliste la vie et la mort : Chaque animal meurt lorsque le nombre de ses mises à jour dépasse 1000. En bon fermier que vous êtes, vous souhaiteriez ajouter un nouvel animal à chaque fois qu'un disparait, mais vous n'êtes pas non plus richissime. Lorsqu'un animal meurt, il y a donc 70% de chance qu'un nouvelle animal apparaisse.

Bien sûr, vous pouvez aussi créer vos propres animaux.

9.2 plantes

Un fermier, ce n'est pas que des animaux. Votre oncle vous l'aurait rabâché sans cesse s'il était encore de ce monde. Une gestion des plantes est important dans une ferme, sinon c'est la famine assuré.

Gérer la flore implique aussi que n'importe quelle case peut maintenant se mettre à jour, la méthode *update()* ne doit donc plus être dans la classe Animal mais directement dans la classe Drawable.

- À chaque mise à jour, une case terre a 25% de chance de devenir de l'herbe.

-Vous pouvez aussi planter du maïs. Si une case herbe est non occupée, elle a 10% de chance de faire pousser du maïs. Quand un épi a poussé, il attire instantanément les cochons qui n'évitent plus l'herbe. Le premier qui arrive sur la case mange l'épi et le sol redevient de la terre.

- Si toute la ferme n'est plus occupée que par de la terre, les animaux meurent et c'est la fin de partie.

Voilà comment fonctionne une ferme. Il est maintenant temps de prendre la relève du vieil O'Brien !