Programmation orientée objets et développement de jeux

Version 0.1

Cédric Donner

03 June 2014

Table des matières

CHAPITRE 1

Programmation orientée objets

1.1 Introduction

La programmation orientée objet est un *paradigme de programmation* informatique qui consiste en la définition et l'assemblage de "briques informatiques", appelées objets. Un objet est une entité que l'on construit par instanciation à partir d'une classe. Une classe est en quelque sorte une « catégorie » ou un « type » d'objets représentant un concept, une idée ou toute entité du monde physique comme une voiture, une personne ou encore un livre. En fait, vous avez déjà rencontré et manipulé des objets en Python puisque toutes les variables existant en Python sont des objets, ainsi que les listes, tuples et dictionnaires. En Python, même les fonctions sont des objets.

L'objectif de ce chapitre est d'apprendre à définir de nouvelles classes d'objets. Il s'agit là d'un sujet relativement ardu, mais vous l'aborderez de manière très progressive, en commençant par définir des classes d'objets très simples, que vous perfectionnerez ensuite. En effet, comme les objets de la vie courante, les objets informatiques peuvent être très simples ou très compliqués. Ils peuvent être composés de différentes parties, qui soient elles-mêmes des objets, ceux-ci étant faits à leur tour d'autres objets plus simples, etc.

Astuce : L'utilisation de classes dans vos programmes vous permettra, entre autres avantages, d'éviter au maximum l'emploi de variables globales. Vous devez savoir en effet que l'utilisation de variables globales comporte des risques, d'autant plus importants que les programmes sont volumineux, parce qu'il est toujours possible que de telles variables soient modifiées, ou même redéfinies, n'importe où dans le corps du programme. Ce risque s'aggrave particulièrement si plusieurs programmeurs différents travaillent sur un même logiciel.

1.2 Concepts de base

1.2.1 Objet = attributs + méthodes

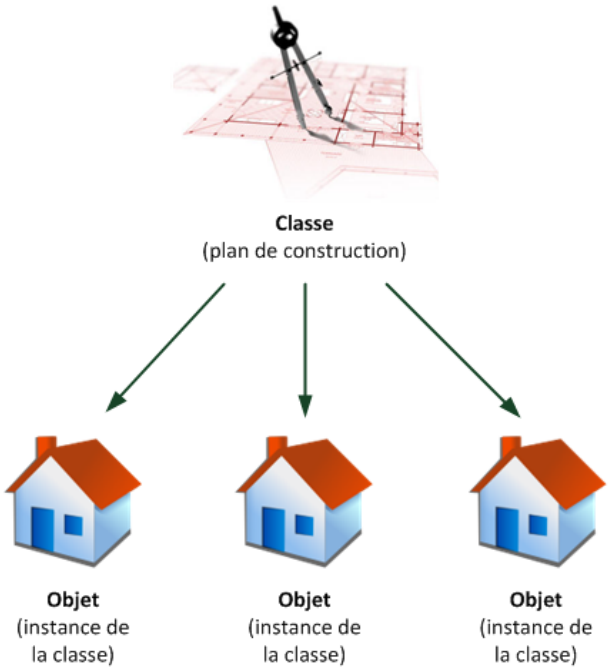
La programmation orientée objet (Object Oriented Programming ou OOP) permet de structurer les logiciels complexes en les organisant comme des ensembles d'objets qui interagissent entre eux et avec le monde extérieur. L'idée de base de la programmation orientée objet consiste à regrouper dans un même ensemble (l'objet), à la fois un certain nombre de données (les attributs d'instance) et les algorithmes destinés à effectuer divers traitements sur ces données (les méthodes) :

Objet = [attributs + méthodes]

Cette façon d'associer dans une même "capsule" les *propriétés* d'un objet et les fonctions qui leur permettent d'interagir avec le monde extérieur, correspond chez les concepteurs de programmes à une volonté de construire des entités informatiques dont le comportement se rapproche du comportement des objets du monde réel.

Les **classes** sont les principaux outils de la programmation orienté objet et consistent en la définition des caractéristiques et des comportements propres à tous les objets d'une même famille. Ainsi, une classe est avant tout une structure de données dans la mesure où il s'agit d'une entité chargée de gérer, classer et stocker des données sous une certaine forme. La principale différence entre une classe et une quelconque structure de données réside dans le fait qu'une classe regroupe des données, sous la forme d'attributs, mais également le moyen de les traiter, sous la forme de méthodes.

En résumé, une classe peut être identifiée à un *moule* utilisé pour créer des objets d'une même famille. Ce moule se compose d'attributs, des données représentant l'état de l'objet, et de méthodes, des opérations applicables aux objets. Un objet, appelés également instance de classe, est dès lors la réalisation concrète d'une classe possédant des caractéristiques qui leur sont propres. Par exemple, une maison est une instance particulière d'un plan la définissant :



1.2.2 Apports de la programmation orientée objets

Encapsulation

Le premier bénéfice de la programmation orientée objet réside dans le fait que les différents objets utilisés peuvent être construits indépendamment les uns des autres (par exemple par des programmeurs différents) sans qu'il n'y ait de risque d'interférence. Ce résultat est obtenu grâce au concept **d'encapsulation** : la fonctionnalité interne de l'objet et les variables qu'il utilise pour effectuer son travail, sont en quelque sorte « enfermées » dans l'objet. Les autres objets et le monde extérieur ne peuvent y avoir accès qu'à travers des procédures bien définies : l'**interface** de l'objet. Ainsi, de l'extérieur, un objet est perçu comme une boîte noire ayant certaines propriétés et un comportement spécifié. Cette boite noire établit une séparation entre la fonctionnalité d'un objet et la manière dont cette fonctionnalité est réellement implémentée dans l'objet :

L'encapsulation consiste donc à garantir l'intégrité des données contenues dans l'objet en s'assurant que son utilisateur ne puisse pas modifier des attributs clés ou manipuler incorrectement l'objet sans passer par le contrôle de l'interface. L'usage d'un objet à travers son interface permet d'en simplifier la manipulation tout en évitant à son utilisateur de devoir savoir faire appel aux mécanismes fondamentaux régissant son fonctionnement.

Un second bénéfice résultant de l'utilisation des classes est la possibilité qu'elles offrent de construire de nouveaux objets à partir d'objets préexistants, et donc de réutiliser des pans entiers d'une programmation déjà écrite (sans toucher

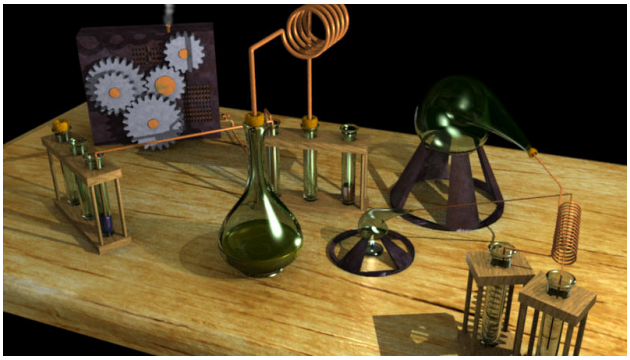


FIGURE 1.1 – Outils de laboratoires : accès direct

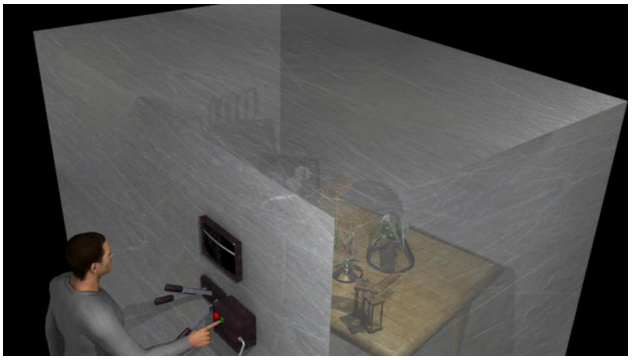
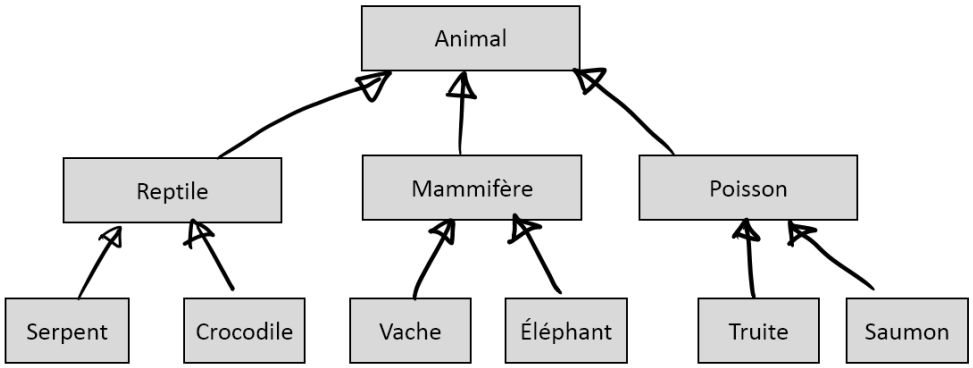


FIGURE 1.2 – Outils de laboratoires : accès par une interface

à celle-ci !), pour en tirer une fonctionnalité nouvelle. Cela est rendu possible grâce aux concepts de dérivation et de polymorphisme :

Héritage

Un second bénéfice résultant de l'utilisation des classes est la possibilité qu'elles offrent de *construire de nouveaux objets* à partir d'objets préexistants, et donc de réutiliser des pans entiers d'une programmation déjà écrite (sans toucher à celle-ci !), pour en tirer une fonctionnalité nouvelle. Cela est rendu possible grâce aux concepts de dérivation et de **polymorphisme** :



Polymorphisme

Le polymorphisme permet d'attribuer des comportements différents à des objets dérivant les uns des autres, ou au même objet ou en fonction d'un certain contexte. En proposant d'utiliser un même nom de méthode pour plusieurs types d'objets différents, le polymorphisme permet une programmation beaucoup plus générique. Le développeur n'a pas à savoir, lorsqu'il programme une méthode, le type précis de l'objet sur lequel la méthode va s'appliquer. Il lui suffit de savoir que cet objet implémentera la méthode.

En savoir plus

Consulter les références suivantes :
– http://www.commentcamarche.net/contents/811-poo-le-polymorphisme

1.3 Apprentissage par l'exemple

1.3.1 Définition d'une classe

Un bon exemple vaut mieux qu'on long discours pour cerner immédiatement les différentes notions de base de la programmation orientée objets. Enfilez vos meilleures lunettes (ou lentilles) et observez attentivement le code suivant qui montre comment définir une nouvelle classe `Recipient` qui dérive de la classe de base `object` :

```
1  class Recipient(object):
2
3  def __init__(self, contenance, no):
4      self.no = no
5      self.contenance = contenance
6      self.volume = 0
7
8  def vider(self):
9      self.volume = 0
10
11 def transferer(self, autre):
12      quantite_transferee = min(autre.contenance - autre.volume, self.volume)
13      autre.volume += quantite_transferee
14      self.volume -= quantite_transferee
15
16 def remplir(self):
17     self.volume = self.contenance
18
19 def __str__(self):
20      return "Récipient no {}, Contenance : {}, Volume : {}".format(self.no,
21                                                                    self.contenance,
22                                                                    self.volume)
23
24 def __repr__(self):
25      return str(self)
```

Analyse du code

Étudiez attentivement le code ci-dessus à l'aide des annotations.

Astuce : Si vous ne comprenez pas tout, c'est normal ... la programmation orientée objets est difficile d'accès au début, mais dès qu'on commence à la maîtriser, c'est un petit paradis de la programmation.


```
1  class Recipient(object):
2
3      def __init__(self, contenance, no):
4          self.no = no
5          self.contenance = contenance
6          self.volume = 0
7
8      def vider(self):
9          self.volume = 0
10
11     def transferer(self, autre):
12         quantite_transferee = min(autre.contenance - autre.volume, self.volume)
13         autre.volume += quantite_transferee
14         self.volume -= quantite_transferee
15
16     def remplir(self, quantite = None):
17         self.volume = self.contenance
18
19     def __str__(self):
20         return "Récipient no {}, Contenance : {}, Volume : {}".format(self.no,
21                                                                       self.contenance,
22                                                                       self.volume)
23
24     def __repr__(self):
25         return str(self)
```

Cette classe va nous permettre d'aider Bruce Willis à désamorcer une bombe ...

1.3.2 Utilisation de la classe

Pour pouvoir utiliser une classe, il faut créer un ou plusieurs objets de cette classe. En termes techniques, on dit qu'on crée une **instance** de la classe ou qu'on **instancie** la classe. Ainsi, après avoir défini la classe à l'aide du mot-clé `class`, on peut créer des récipients différents.

```
>>> r1 = Recipient(no = 1, contenance = 5)
>>> r2 = Recipient(no = 2, contenance = 3)
```

À l'aide de ces deux lignes, ont vient de créer deux objets concrets (instances) de la classe `Recipient` (on pourrait dire du *type* `Recipient`). En effet, notre classe `Recipient` constitue un nouveau type de données utilisable dans notre programme.

```
>>> r2.remplir() # remplir le recipient r2
>>> r1.transferer(r2) # transfère le contenu de r2 dans r1
>>> r2.vider() # vider le récipient r2
>>> r2.volume
0
>>> r1.volume
3
...
>>> r2.volume == 2 # ce que l'on devrait obtenir à la fin ...
True
```

Récréation

Dans *Die Hard 3*, Bruce Willis a besoin de 4 "gallons" d'eau pour désamorcer une bombe, mais il ne dispose que d'un récipient de contenant 3 gallons et un autre de contenance 5 gallons. Comment doit-il s'y prendre ?

Astuce : Regarder la séquence du film https://www.youtube.com/watch?v=BVtQNK_ZUJg

Utilisez la classe `Recipient` définie ci-dessus pour écrire un programme qui permet d'avoir 4 gallons dans le grand récipient.

Contraintes

Les récipients ne sont pas gradués. On peut donc uniquement faire les opérations suivantes avec les récipients :

– Remplir le récipient `r1` avec

```
r1.remplir()
```

– Transférer le contenu du récipient `r1` dans le récipient `r2`. Uniquement le volume encore disponible dans le récipient `r2` sera transféré, le reste demeure dans le récipient de départ `r1`.

```
r1.transferer(r2)
```

– Vider le récipient avec

```
r1.vider()
```

Astuce : Si vous ne parvenez pas à la solution, faites une recherche sur Google avec la requête

```
Die Hard 3 jug riddle
```

Code de base

```
# On commence par créer les instances de la classe Recipient en
# spécifiant la capacité
r1 = Recipient(no = 1, contenance = 5)
r2 = Recipient(no = 2, contenance = 3)

# Procédure à compléter  ...



# La bombe ne peut être désamorcée que si le volume de r1 vaut 4
# gallons
assert(r1.volume == 4)
```

CHAPITRE 2

Jeux et POO

2.1 Encore des objets, rien que des objets

2.1.1 Introduction

Dans la vie de tous les jours, vous êtes entouré par une quantité d'objets. Parce que le logiciel est un modèle qui reflète souvent une certaine réalité, il est naturel d'introduire des objets dans l'informatique. C'est ce qu'on appelle la programmation orientée objet (POO). Ce **paradigme de programmation** s'est imposé depuis deux décennies dans le génie logiciel, à tel point qu'il est impensable aujourd'hui de développer un projet informatique sans passer en n'utilisant pas la programmation orientée objets. Dans ce chapitre, vous allez apprendre à maitriser les concepts principaux de cette manière de penser et de programmer en développant quelques jeux célèbres.

Sans le savoir, vous avez déjà rencontré plusieurs objets dans votre vie de programmeur/se : les chaines de caractères, listes, dictionnaires et tuples en sont de bons exemples.

2.1.2 Présentation de l'IDE TigerJython

La vidéo suivante présente l'environnement de développer *TigerJython* et les différents éléments son interface graphique.

2.1.3 `JGameGrid` : notre espace jeux

Sans POO, il n'est pas possible de créer un jeu d'ordinateur sans se prendre la tête, car le concepteur de jeux va forcément traiter les personnages du jeu et les autres objets du plateau de jeu comme des objets. Le plateau de jeu est une fenêtre d'écran rectangulaire modélisée par la classe `GameGrid` de la Bibliothèque de jeux `JGameGrid`. Avec un appel à la fonction `makeGameGrid()`, on construit une *instance* (un exemplaire) de la classe `GameGrid` et la méthode `show()` permet d'afficher la fenêtre à l'écran. Il est possible de personnaliser l'apparence de la fenêtre de jeu avec des paramètres appropriés

Création de la grille de jeu

Tous les programmes utilisant `gamegrid` partagent certains ingrédients de base que montrent le code ci-dessous qui se contente de créer la grille de jeux et de l'afficher dans une fenêtre.

Résumé

Avec `makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)`, on peut afficher une fenêtre dans laquelle on pourra développer nos jeux.
– Les trois premiers paramètres indiquent que la fenêtre va contenir une grille de 10×10 cellules carrées qui font 60 pixels de côté.
– Le paramètre `Color.red` crée le grillage rouge qui montre la grille pendant la phase de développement mais il est possible de le supprimer par la suite.
– Le fichier image `sprites/town.jpg` correspond au fond d'écran sur lequel va se dérouler le jeu
– Le dernier paramètre gère l'affichage de la barre de navigation qui n'est pas nécessaire dans cet exemple

Exécuter le code

Pour exécuter les codes Python présentés dans ce chapitre, il faut les coller dans l'environnement *TigerJython* qui est un Python légèrement modifié suivant la version 2.7 du langage.

Pour vous assurer que tout fonctionne correctement, essayez d'exécuter le code suivant dans *TigerJython* :

```
from gamegrid import *

makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)
show()
```

Memo

Les méthodes de la classe `GameGrid` sont mises à ta disposition par la fonction `makeGameGrid()`. Il est cependant également possible de créer une instance de `GameGrid` et d'appeler ses méthodes en utilisant l'opérateur point.

```
from gamegrid import *

gg = GameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)
gg.show()
```

– La fenêtre de jeu est faite de cellules carrées de 60 pixels de côté.
– Il y a 10 cellules horizontales et 20 cellules verticales
– Puisque les lignes de la grilles sont affichées également tout en bas et tout à droite, la fenêtre a une taille de 601 x 601 pixels. Cela correspondant à la taille minimale de l'image d'arrière-plan
– Le dernier paramètre booléen détermine si une barre de navigation apparaît.

Création de nouveaux personnages dérivés de la classe `Actor`

Les jeux que nous allons développer font intervenir des personnages, plus précisément des *sprites* qui sont appelés *acteurs* dans la philosophie de la GameGrid. Pour créer un nouveau personnage, il faut donc créer une nouvelle classe qui dérive de la classe `Actor`.

La vidéo ci-dessous montre comment créer une nouvelle classe `Alien` qui dérive de la classe `Actor` pour représenter les aliens qui vont tomber du ciel dans notre jeu qui va évoluer en une sorte de *Space Invader* très simplifié.

Code à étudier

```
1 from gamegrid import *
2
3 # ---------------- class Alien ----------------
4 class Alien(Actor):
```

```
    def __init__(self):
        Actor.__init__(self, "sprites/alien.png")

    def act(self):
        self.move()

makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)
spin = Alien() # object creation, many instances can be created
urix = Alien()
addActor(spin, Location(2, 0), 90)
addActor(urix, Location(5, 0), 90)
show()
doRun()
```

Analyse du code

Résumé

Voici un résumé de l'analyse de code effectuée dans la vidéo. Prenez le temps de bien comprendre chaque élément de ce code :

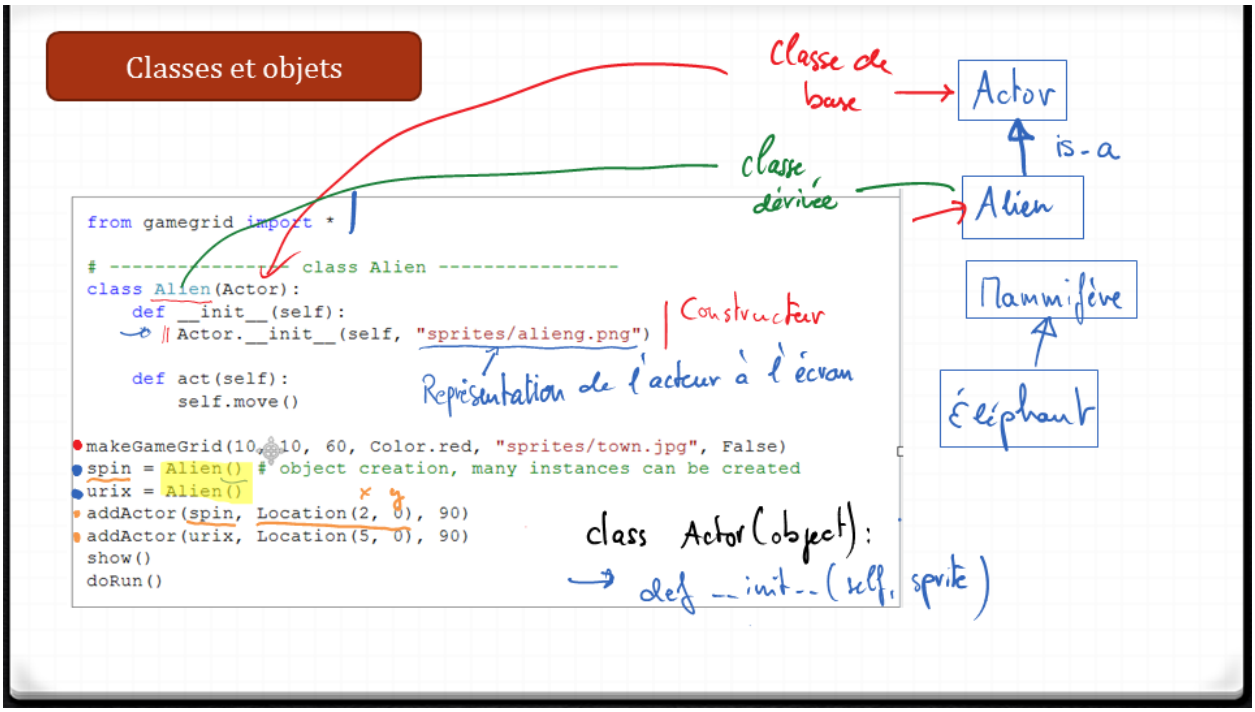


FIGURE 2.1 – Création d'une classe `Alien` qui dérive de `Actor`

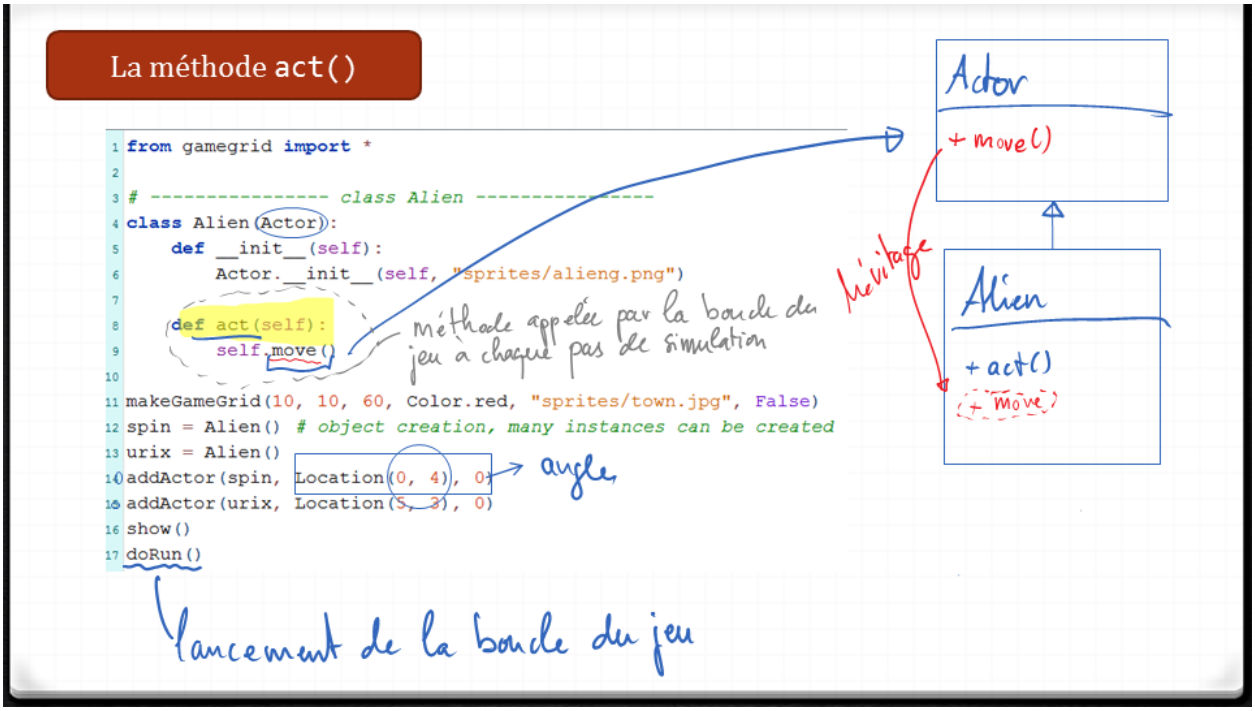
Bien comprendre comment fonctionne la méthode `act()` des acteurs

La méthode `act()` définie dans l'interface de la classe `Actor` doit être réimplémentée (surchargée) dans les classes qui dérivent de la casse `Actor`. Tout acteur personnalisé doit donc définir cette méthode `act()` qui spécifie ce que doit faire l'acteur à chaque cycle de la simulation (boucle du jeu). Cette méthode sera donc appelée par exemple toutes les 30 millisecondes pour déplacer l'acteur à son prochain emplacement dans la grille.

La méthode `move()` appelée par la méthode `act()` est une méthode d'instance qui n'est cependant pas définie dans la classe `Alien`. Celle-ci est définie plutôt dans la classe de base `Actor` et héritée par la classe dérivée `Alien` (cf. figure *fig-explication-methode-act*)

Résumé de la vidéo

Voici de quoi comprendre le code analysé dans la vidéo :



Space Invader Light

Nous allons compléter notre petit programme pour pouvoir éliminer les aliens qui tombent en leur cliquant dessus. Pour cela, il faut créer un gestionnaire d'événement, matérialisé dans notre code par la fonction `pressCallback(e)`. Cette fonction est connectée aux événements de clic sur la souris (bouton gauche) par le dernier paramètre `mousePressed = pressCallback` à la ligne 20 :

```
makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False,
         mousePressed = pressCallback)
```

Code Python

```
1  from gamegrid import *
2  import random
3
4  # ---------- class Alien ----------
5  class Alien(Actor):
6      def __init__(self):
7          Actor.__init__(self, "sprites/alien.png")
```

```
8
9       def act(self):
10          self.move()
11
12  def pressCallback(e):
13      location = toLocationInGrid(e.getX(), e.getY())
14      actor = getOneActorAt(location)
15      if actor != None:
16          removeActor(actor)
17      refresh()
18
19  makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False,
20               mousePressed = pressCallback)
21  setSimulationPeriod(800)
22  show()
23  doRun()
24
25  while not isDisposed():
26      alien = Alien()
27      addActor(alien, Location(random.randint(0, 9), 0), 90)
28      delay(1000)
```

Nouveautés

- Un gestionnaire d'événements est une **fonction** qui prend un seul paramètre `e` qui contient les informations contenant l'événement qui a été généré (par exemple un clic de souris). Dans notre code, c'est la fonction `pressCallback` qui fait office de gestionnaire d'événements pour les clics de souris.
  - `e.getX()` : coordonnées `x` du clic `(0, 0)` étant tout en haut à gauche de la fenêtre (pixels)
  - `e.getY()` : coordonnées `y` du clic `(0, 0)` étant tout en haut à gauche de la fenêtre (pixels)
  - Les coordonnée `(0, 0)` correspond au coin supérieur gauche de l'espace contenu dans la fenêtre du jeu
- `toLocationInGrid()` transforme les coordonnées exprimées sous forme de pixels en une adresse de cellule dans la grille de jeu. Si les cases de la grille font par exemple 60×60‘ pixels, on aura

```
>>> toLocationInGrid(32, 25)
(0, 0)
>>> toLocationInGrid(61, 25)
(1, 0)
```

- `setSimulationPeriod(800)` fait en sorte que la méthode `act()` des acteurs soit appelée toutes les 800 ms

2.1.4 Devoirs

Page d'origine

http://tigerjython.ch/index.php?inhalt_links=navigation.inc.php&inhalt_mitte=gamegrid/objekte.inc.php

1. Générer une image avec un éditeur d'images ou prendre une image libre de droits sur Internet et la placer en image de fond du jeu.

   Conseils
   - Créer un dossier `sprites` dans le dossier dans lequel se trouve le fichier `tigerjython.jar`
   - Déposer votre image dans ce dossier `sprites`
   - modifier l'appel à la fonction `makeGameGrid` pour charger votre fichier

   Remarque : il est également possible de charger une image se trouvant n'importe où sur le disque mais il faudra alors indiquer le chemin absolu complet de ce fichier.

2. Ajouter une barre d'état de 30 pixels de hauteur qui indique combien d'aliens ont pu atterrir dans la ville malgré la vigilence du joueur.

Conseils
– utiliser `addStatusBar(<nbre_pixels>)` pour ajouter la barre d'état à la fenêtre du jeu.
– utiliser `setStatusText(chaine)` pour afficher la chaine `chaine` dans la barre d'état.

---

3. Les aliens qui ont atterri dans la ville ne doivent pas simplement disparaître mais doivent, une fois atterri dans la ville, prendre une autre forme (par exemple `sprites/alien_1.png`) et garder leur position.

Conseil
– Il faut modifier la méthode `act()` des aliens qui est appelée à chaque cycle du jeu pour les faire bouger. À l'intérieur de cette méthode, `self` fait référence à l'acteur courant.
– Consulter la documentation de *GameGrid* (en allemand pour le moment : http://tigerjython.ch/index.php?inhalt_links=navigation.inc.php&inhalt_mitte=gamegrid/gamegriddoc.inc.php). Cette page présente surtout les méthodes des classes `GameGrid`, `Actor`, `Location` et `GGBackground`. Ce sont surtout les classes `GameGrid`, `Actor` et `Location` qui nous intéressent pour le moment.
– Dans cette documentation, on apprend que la classe `Alien` définit les méthodes `getX()`, `getY()` qui permet de déterminer les coordonnées x et y d'un acteur dans la grille de jeu.
– Il est aussi possible de déterminer si un acteur se trouve dans la dernière rangée de la grille à l'aide de la méthode `Actor.isMoveValid()` qui permet de savoir si le prochain appel de la méthode `move()` de cet acteur va le faire sortir de la grille.
– La méthode `Actor.removeSelf()` permet de supprimer du jeu l'acteur sur lequel la méthode est appelée. Donc, dans la méthode `Alien.act()`, on peut faire appel à cette méthode avec

```
def act(self):
    # supprimer l'acteur
    if condition():
        self.removeSelf()
```

---

4. Lorsque les aliens atterrissent, ils communiquent à leur vaisseau spatial le numéro de la colonne dans laquelle ils ont atterri. Celui-ci ne va plus larguer d'alien dans ces colonnes, mais uniquement dans les colonnes dans lesquelles aucun alien n'a encore atterri.

Conseils
– Consulter la documentation du module `random` de Python, en particulier pour la méthode `random.choice`

---

5. Faire preuve de créativité pour ajouter d'autres règles au jeu.

Suggestions
(a) Compter le nombre de points pour chaque alien tué et afficher le score dans la barre d'état
(b) Accélérer la chute des aliens et/ou réduire le délai de parachutage des aliens lorsque le score dépasse certains seuils
(c) ...

Corrigé complet de tous les exercices

Voici le code obtenu après correction de tous les exercices. Il constitue donc un code fonctionnel implémentant toutes les fonctionnalités demandées dans les exercices.

```
from gamegrid import *
import random

# ---------------- class Alien ----------------
class Alien(Actor):

    # variable de classe
    nb_alien_atterri = 0

    def __init__(self):
        Actor.__init__(self, "sprites/alien_0.gif")

    def act(self):
        self.move()

        if self.getY() == 9:
            landedAlien = LandedAlien()
            addActor(landedAlien, Location(self.getX(), 9))

            try:
                free_columns.remove(self.getX())
            except:
                pass

            Alien.nb_alien_atterri += 1
            setStatusText("Nombre d'aliens qui ont atterri : " + str(Alien.nb_alien_atterri))

class LandedAlien(Actor):

    def __init__(self):
        Actor.__init__(self, "sprites/alien_1.gif")


def pressCallback(e):
    location = toLocationInGrid(e.getX(), e.getY())
    actor = getOneActorAt(location)
    if actor != None:
        removeActor(actor)
    refresh()

makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False,
             mousePressed = pressCallback)
setSimulationPeriod(100)
addStatusBar(30)
setStatusText("Nombre d'aliens qui ont atterri : " + str(0))
show()
doRun()

free_columns = list(range(10))

while not isDisposed():
    alien = Alien()
    try:
        addActor(alien, Location(random.choice(free_columns), 0), 90)
    except:
        setStatusText("Game over !!!")
    delay(1000)
```

2.2 Classes et objets

2.2.1 Récapitulation des notions de base de la POO

Observez attentivement le code ci-dessous et répondre aux questions. Toutes les questions posées sont vraiment essentielles et donc des questions types qui peuvent être posées lors d'un oral de BAC.

```
1  from gamegrid import *
2
3  # ---------------- classe Animal ----------------
4  class Animal():
5
6      def __init__(self, imgPath):
7          self.imagePath = imgPath
8
9
10     def showMe(self, x, y):
11          bg.drawImage(self.imagePath, x, y)
12
13
14 def pressCallback(e):
15     myAnimal = Animal("sprites/animal.gif")
16     myAnimal.showMe(e.getX(), e.getY())
17
18 makeGameGrid(600, 600, 1, False, mousePressed = pressCallback)
19 setBgColor(Color.green)
20 show()
21 doRun()
22 bg = getBg()
```

Analyse de code

1. Décrire le rôle de la fonction `__init__` aux lignes 6 et 7

   Corrigé
   La fonction `__init__` est le constructeur de la classe `Animal`. Cette fonction est appelée automatiquement à fois que l'on crée un animal avec

   ```
   mon_animal = Animal("sprites/example.png")
   ```

   ---

2. Décrire précisément ce qui se passe à la ligne 7

   Corrigé
   Cette ligne crée la variable d'instance `self.imagePath` et l'initialise avec le contenu de la variable locale `igmPath`.

   ---

3. Que représente le premier paramètre `self` dans la définition des méthodes d'instance ?

   Corrigé
   Le paramètre `self` est propre à toutes les méthodes d'instance et doit toujours se trouver en première position. Il s'agit d'une référence vers l'instance concrète sur laquelle la méthode a été invoquée.
   Lors de l'invocation de la méthode avec

```
mon_animal.showMe(10, 20)
```

on ne renseigne pas ce paramètre `self` car Python s'en charge pour nous en transformant notre appel dans le code suivant avant de l'exécuter :

```
Animal.showMe(mon_animal, 10, 20)
```

---

4. À quoi sert la fonction `pressCallback(e)` définie aux lignes 14 à 16 ?

Corrigé
Cette fonction est un **gestionnaire d'événement** (*Event handler* en anglais). Elle sera appelée à par le système du jeu à chaque fois qu'un événement de type `MousePressed` est généré par le système.
C'est uniquement à la ligne 18

```
makeGameGrid(600, 600, 1, False, mousePressed = pressCallback)
```

que notre fonction `pressCallback` est "connectée" à l'événemnt `mousePressed`. Ce qui se passe à la ligne 18 est très nouveau : on passe à la fonction `makeGameGrid` la fonction `pressCallback` en guise de paramètre. Notez bien que l'on n'a pas écrit `mousePressed = pressCallback()` mais bien `mousePressed = pressCallback` sans appeler la fonction `pressCallback` avec des parenthèses `()`.

---

5. Que représente le paramètre `e` de la fonctoin `pressCallback(e)` ?

Corrigé
Il s'agit d'un objet représentant l'événement qui a déclenché l'appel de `pressCallback`. Cet objet `e` contient des informations sur l'événement généré par le clic de souris, en particulier les coordonnées du clic récupérables avec `e.getX()` et `e.getY()`.

---

6. Décrire précisément ce qui se passe à la ligne 16 ?

Corrigé
En gros, on crée une instance de la classe `Animal` à la ligne 15 que l'on affiche à la ligne 16 à l'emplacement du clic de la souris.

---

7. Expliquer ce que fait globalement ce code Python ?

Corrigé
Globalement, le programme affiche des petits animaux lorsqu'on clique dans l'espace de jeu. Le coin supérieur gauche du rectangle contenant le sprite de l'animal correspondra aux coordonnées du clic de la souris.

2.2.2 Héritage

L'héritage est une des propriétés les plus utiles et fondamentales dans la POO. Ce mécanisme permet de réutiliser du code défini dans d'autres classes par dérivation. Observer ce code en répondre aux questions posées :

Appel du construteur de la classe de base

Dans la version 2.7 de Python utilisée par TigerJython, on peut écrire

```
super(Pet, self).__init__(self, imgPath)
```

pour appeler le constructeur de la classe de base de `Pet` pour éviter d'y faire référence explicitement comme le fait notre code avec

```
Animal.__init__(self, imgPath)
```

Dans Python 3, il est possible de se contenter de

```
super().__init__(self, imgPath)
```

ce qui est nettement plus élégant

```
from gamegrid import *
# Une des forces de TigerJython est qu'il permet d'utiliser
# les bibliothèques Java
from java.awt import Point

# ---------------- classe Animal ----------------
class Animal():

    def __init__(self, imgPath):
        self.imagePath = imgPath


    def showMe(self, x, y):
         bg.drawImage(self.imagePath, x, y)

# ---------------- classe Pet ----------------
class Pet(Animal):   # Derived from Animal

    def __init__(self, imgPath, name):
        Animal.__init__(self, imgPath)
        self.name = name

    def tell(self, x, y): # Additional method
        bg.drawText(self.name, Point(x, y))

makeGameGrid(600, 600, 1, False)
setBgColor(Color.green)
show()
doRun()
bg = getBg()
bg.setPaintColor(Color.black)

for i in range(5):
    myPet = Pet("sprites/pet.gif", "Trixi")
    myPet.showMe(50 + 100 * i, 100)
    myPet.tell(72 + 100 * i, 145)
```

Questions

1. Pourquoi met-on `Animal` entre parenthèses après `class Pet` dans la définition de la casse `Pet` ?

Corrigé
Pour indiquer que la classe `Pet` est basée sur la classe `Animal`. Autrement dit, la classe `Pet` hérite de toutes les propriétés (variables d'instances) et méthodes de la classe `Animal`.

---

2. Décrire précisément ce que fait la ligne 20

Corrigé
La ligne 20

```
Animal.__init__(self, imgPath)
```

appelle explicitement le constructeur de la classe parent pour lui déléguer l'initialisation des attributs d'instances définis dans la classe `Animal`.

Astuce : Il faut toujours déléguer l'initialisation des variables d'instances définies dans la classe de base au construteur de la classe de base.

---

3. Décrire ce que fait le programme globalement

Corrigé
En général, un bon programmeur peut prendre le code de quelqu'un d'autre et savoir ce qu'il va faire sans l'exécuter. Le commun des mortels exécutent le programme et s'étonnent lorsque ça ne fonctionne pas comme ils pensaient ...

---

4. Dessiner le diagramme de classes de `Animals` et `Pet`

Corrigé

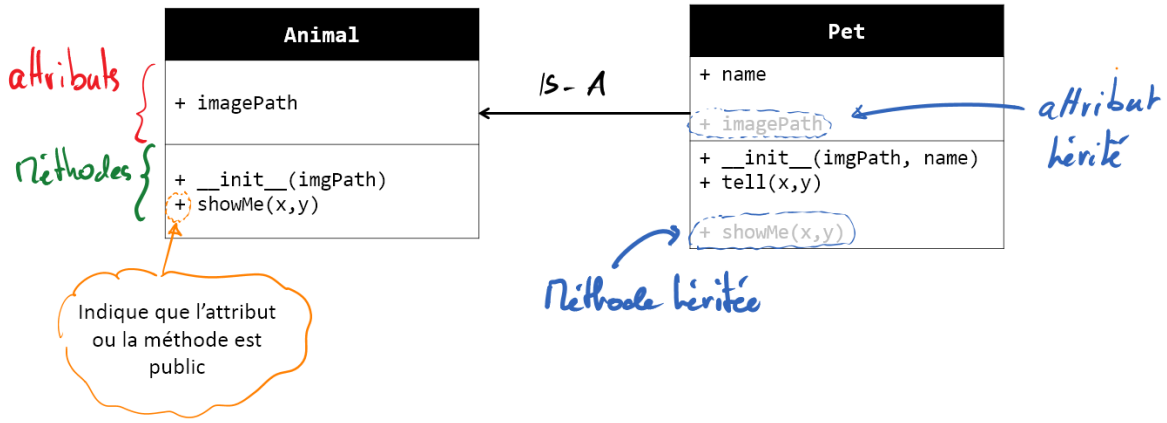


FIGURE 2.2 – Diagramme de classes montrant la classe `Pet` dérivée de la classe `Animal`

À retenir

On peut appeler sans problème la méthode `myPet.showMe()` alors même que `showMe()` n'est pas définie dans la classe `Pet`, car un animal de compagnie est bien un animal et dispose donc des mêmes comportements. On appelle cette relation des classes `Pet` et `Animal` une relation **Est-Un(e)** (*IS-A* en anglais), car un `Pet` *est un* `Animal`.

Pour que `DerivedClass` dérive de la classe `BaseClass`, il suffit d'indiquer `(BaseClass)` entre parenthèses dans la définition de `DerivedClass` :

```
class BaseClass:

    # définition de la classe de base
    pass

class DerivedClass(BaseClass):

    # définition de la classe dérivée
    pass
```

En Python 3, par convention, les classes qui n'ont pas de classe de base dérivent de la classe `object` et il est considéré comme une bonne pratique de l'indiquer explicitement :

```
# classe qui ne dérive d'aucune autre classe
class MaClasse(object):

    # définition de la classe
    pass
```

Le langage Python permet l'**héritage multiple** (*Mutliple inheritance* en anglais), ce qui permet de dériver une classe à partir de plusieurs classes de base :

```
# classe qui ne dérive d'aucune autre classe
class MaClasse(BaseClass1, BaseClass2):

    # définition de la classe
    pass
```

Hiérarchie de classes

Étudier attentivement le code suivant et répondre aux questions :

```
from gamegrid import *
from java.awt import Point

# ---------------- classe Animal ----------------
class Animal():

    def __init__(self, imgPath):
        self.imagePath = imgPath


    def showMe(self, x, y):
         bg.drawImage(self.imagePath, x, y)

# ---------------- classe Pet ----------------
class Pet(Animal):

    def __init__(self, imgPath, name):
        Animal.__init__(self, imgPath)
        self.name = name

    def tell(self, x, y):
        bg.drawText(self.name, Point(x, y))

# ---------------- classe Dog ----------------
```

```
class Dog(Pet):

    def __init__(self, imgPath, name):
        Pet.__init__(self, imgPath, name)

    def tell(self, x, y): # Overriding
        bg.setPaintColor(Color.blue)
        bg.drawText(self.name + " tells 'Waoh'", Point(x, y))

# ---------------- classe Cat ----------------
class Cat(Pet):

    def __init__(self, imgPath, name):
        Pet.__init__(self, imgPath, name)

    def tell(self, x, y): # Overriding
        bg.setPaintColor(Color.gray)
        bg.drawText(self.name + "  tells 'Meow'", Point(x, y))

makeGameGrid(600, 600, 1, False)
setBgColor(Color.green)
show()
doRun()
bg = getBg()

alex = Dog("sprites/dog.gif", "Alex")
alex.showMe(100, 100)
alex.tell(200, 130)

rex = Dog("sprites/dog.gif", "Rex")
rex.showMe(100, 300)
rex.tell(200, 330)

xara = Cat("sprites/cat.gif", "Xara")
xara.showMe(100, 500)
xara.tell(200, 530)
```

Questions

1. Dessiner le diagramme de classes de `Animal`, `Pet`, `Cat`, `Dog`

Corrigé

---

2. Modifier les classes `Dog` et `Cat` pour qu'elles chargent automatiquement le bon sprite (la bonne image représentative) sans devoir le spécifier dans le construteur

---

Corrigé
Il suffit de modifier le construteur des classes `Dog` et `Cat`. Par exemple, pour la classe `Dog`, en supposant que le sprite voulu soit `sprites/dog.png`, il apporter les modifications suivantes :

```
class Dog(Pet):

    def __init__(self, name):
        Pet.__init__(self, imgPath="dog.png")
        self.name = name
```

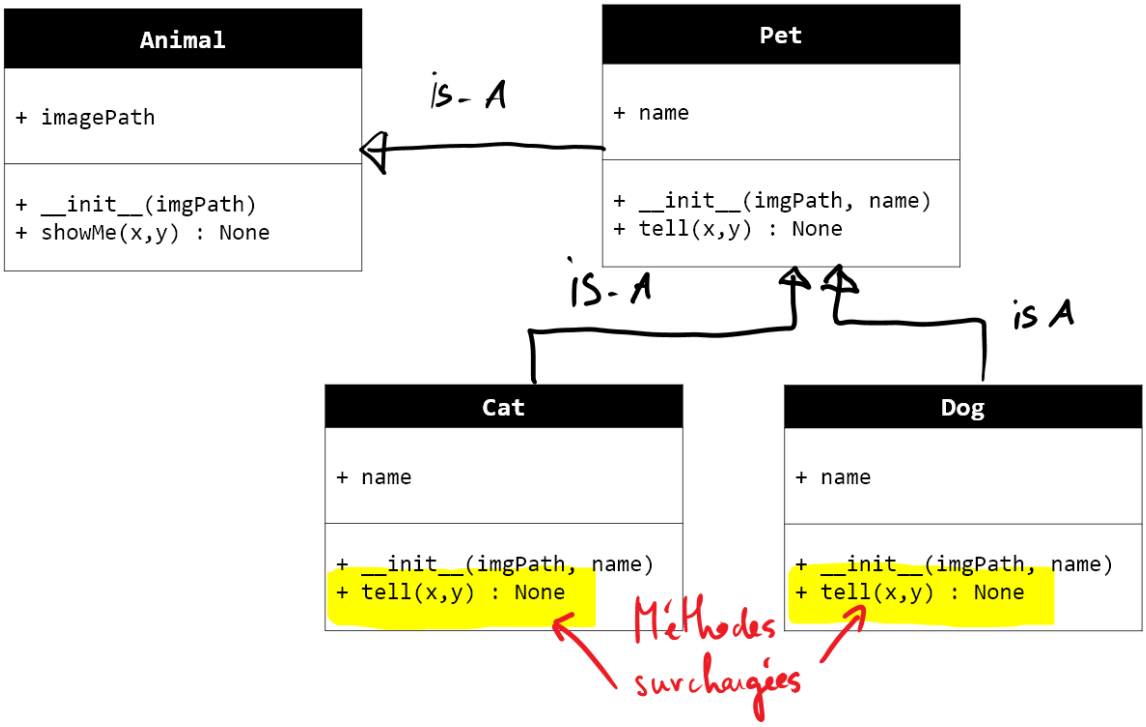


FIGURE 2.3 – Diagramme de classes montrant la classe `Pet` dérivée de la classe `Animal`

À retenir

1. Lorsqu'un méthode est redéfinie à plusieurs endroits dans une hiérachie de classes, le polymorphisme permet d'appeler la bonne version de la méthode. En Python, le polymorphisme est logique puisque le type des données est déterminé de manière dynamique lors de l'exécution. Toutefois, dans les langages à typage statique tels que Java ou C++, le mécanisme de polymorphisme est bien moins trivial.
2. Le polymorphisme pratiqué par Python est parfois appelé *Duck-Typing* selon la citation de James Whitcomb Riley (1849 - 1916) :

   > Si je vois un oiseau qui marche comme un canard, nage comme un canard et qui cancane comme un canard, alors il s'agit d'un canard

3. Il arrive qu'une méthode soit définie dans la classe de base mais qu'elle ne doive rien faire du tout à part être redéfinie dans les classes dérivées. On parle alors de **méthode virtuelle** définie dans une **classe abstraite**.

   En Python, on spécifie qu'une méthode est virtuelle et ne doit donc pas être appelée en levant l'exception `NotImplementedError`

```
class AbstractClass(object):

    def __init__(self):
        pass

    def VirtualMethod(self):
        raise NotImplementedError
```

Es gibt Fälle, wo eine überschriebene Methode in der Basisklasse zwar definiert ist, aber nichts bewirken soll. Dies erreicht man entweder mit einem sofortigen return oder mit der leeren Anweisung pass.

2.2.3 Polymorphisme

Le polymorphisme consiste à **surcharger** (*override* en anglais) les méthodes de la classe de base dans les classes dérivées. Ici, en l'occurrence, on utilise ce mécanisme pour surcharger la méthode `tell` dans les classes `Dog` et `Cat` :

```
from gamegrid import *
from soundsystem import *

# ---------------- classe Animal ----------------
class Animal():

    def __init__(self, imgPath):
        self.imagePath = imgPath


    def showMe(self, x, y):
         bg.drawImage(self.imagePath, x, y)

# ---------------- classe Pet ----------------
class Pet(Animal):

    def __init__(self, imgPath, name):
        Animal.__init__(self, imgPath)
        self.name = name

    def tell(self, x, y):
        bg.drawText(self.name, Point(x, y))

# ---------------- classe Dog ----------------
class Dog(Pet):

    def __init__(self, imgPath, name):
        Pet.__init__(self, imgPath, name)
        self.name = name

    def tell(self, x, y): # Overridden
        Pet.tell(self, x, y)
        openSoundPlayer("wav/dog.wav")
        play()

# ---------------- classe Cat ----------------
class Cat(Pet):

    def __init__(self, imgPath, name):
        Pet.__init__(self, imgPath, name)
        self.name = name

    def tell(self, x, y): # Overridden
        Pet.tell(self, x, y)
        openSoundPlayer("wav/cat.wav")
        play()


makeGameGrid(600, 600, 1, False)
setBgColor(Color.green)
show()
doRun()
```

```
bg = getBg()

pets = [Dog("sprites/dog.gif", "Alex"),
           Dog("sprites/dog.gif", "Rex"),
           Cat("sprites/cat.gif", "Xara")]

y = 100
for pet in pets:
    animal.showMe(100, y)
    animal.tell(200, y + 30)     # Which tell()????
    y = y + 200
    delay(1000)
```

2.2.4 Toute la puissance du polymorphisme

Il est possible d'obtenir exactement le même résultat avec moins de code. En effet, la seule différence entre les méthodes `Dog.tell` et `Cat.tell` est le nom du fichier WAV joué. Dès lors, on peut éviter de surcharger cette méthode dans les classes filles et faire référence, dans la classe de base `Pet`, à une variable d'instance qui est définie dans la classe de base `Pet` mais qui sera spécifiée de manière différente uniquement dans le construteur des classes dérivées `Cat` et `Dog`.

Observez bien le code ci-dessous pour comprendre comment cela fonctionne :

```
from gamegrid import *
from soundsystem import *

# ---------------- classe Animal ----------------
class Animal():

    def __init__(self, imgPath):
        self.imagePath = imgPath


    def showMe(self, x, y):
         bg.drawImage(self.imagePath, x, y)

# ---------------- classe Pet ----------------
class Pet(Animal):

    def __init__(self, imgPath, name):
        Animal.__init__(self, imgPath)
        self.name = name
        self.sound = None

    def tell(self, x, y):
        bg.drawText(self.name, Point(x, y))
        openSoundPlayer(self.sound)
        play()

# ---------------- classe Dog ----------------
class Dog(Pet):

    def __init__(self, imgPath, name):
        Pet.__init__(self, imgPath, name)
        self.name = name
        self.sound = "wav/dog.wav"

```

```
35
36  # ---------------- classe Cat ----------------
37  class Cat(Pet):
38
39      def __init__(self, imgPath, name):
40          Pet.__init__(self, imgPath, name)
41          self.name = name
42          self.sound = "wav/cat.wav"
43
44
45  makeGameGrid(600, 600, 1, False)
46  setBgColor(Color.green)
47  show()
48  doRun()
49  bg = getBg()
50
51  animals = [Dog("sprites/dog.gif", "Alex"),
52             Dog("sprites/dog.gif", "Rex"),
53             Cat("sprites/cat.gif", "Xara")]
54
55  y = 100
56  for animal in animals:
57      animal.showMe(100, y)
58      animal.tell(200, y + 30)     # Which tell()????
59      y = y + 200
60      delay(1000)
```

Modifications effectuées

Lorsqu'on a deux versions différentes d'un même fichier, on peut toujours observer rapidement les ajouts ou suppression du deuxième fichier par rapport au fichier original avec la commande

```
diff fichier1.py fichier2.py
```

qui donne le résultat suivant

```
19a20
>          self.sound = None
22a24,25
>          openSoundPlayer(self.sound)
>          play()
30,34c33,34
<
<      def tell(self, x, y): # Overridden
<          Pet.tell(self, x, y)
<          openSoundPlayer("wav/dog.wav")
<          play()
---
>          self.sound = "wav/dog.wav"
>
42,46c42
<
<      def tell(self, x, y): # Overridden
<          Pet.tell(self, x, y)
<          openSoundPlayer("wav/cat.wav")
<          play()
---
```

```
>          self.sound = "wav/cat.wav"
64c60
<     delay(1000)
---
>     delay(1000)
```

2.3 Variables et méthodes de classe

Jusqu'à présent, nous avons surtout utilisé des variables et méthodes d'instance.
– Les variables d'instance sont propres à chaque instance de la classe et qui maintenent chacune de ces variables dans des emplacements mémtoires différents.
– Les méthodes d'instance sont des méthodes qui accèdent à des variables d'instances et doivent impérativement débuter par le paramètre `self`.

2.3.1 Variables de classe (variables statiques)

Il arrive parfois que toutes les instances doivent partager certaines variables qui ne concernent aucune instance particulière mais la classe dans son ensemble. Nous avions déjà utilisé ceci pour compter le nombre d'aliens qui ont atterri dans le code *code-space_invader_corrige* dans la classe `Alien` :

```
1  class Alien(Actor):
2
3  # variable de classe partagée par toutes les instances de la classe
4  nb_alien_atterri = 0
5
6  def __init__(self):
7      Actor.__init__(self, "sprites/alien_0.gif")
```

Syntaxe

Au contraire des variables d'instance qui sont définies dans le constructeur avec la syntaxe `self.variable_instance`, les variables de classe sont définies en dehors de toute méthode, au début de la définition sans utiliser le `self` :

```
class MaClasse(object):

    # définition des variables de classe
    variable_de_classe0 = 0
    variable_de_classe1 = "salut"

    def __init__(self):

        # définition des variables d'instance
        self.variable_instance1 = "truc"
        self.variable_instance2 = "machin"
```

Pour accéder aux variables de classe, il n'est pas nécessaire de créer une instance de la classe : suffit d'utiliser la notation `NomClasse.variable` :

```
>>> MaClasse.variable_de_classe0
0
>>> MaClasse.variable_de_classe1 = "ohé!"
```

```
>>> MaClasse.variable_de_classe1
'ohé!'
```

Exemple

Il est par exemple possible d'utiliser des variables de classe pour regrouper des variables dans une classe au lieu de les laisser polluer l'espace de noms global. Il n'est pas nécessaire de créer une instance de la classe pour accéder aux variables de classes.

```
import math

# ---------------- class Physics ----------------
class Physics():
    # Avagadro constant [mol-1]
    N_AVAGADRO = 6.0221419947e23
    # Boltzmann constant [J K-1]
    K_BOLTZMANN = 1.380650324e-23
    # Planck constant [J s]
    H_PLANCK = 6.6260687652e-34;
    # Speed of light in vacuo [m s-1]
    C_LIGHT = 2.99792458e8
    # Molar gas constant [K-1 mol-1]
    R_GAS = 8.31447215
    # Faraday constant [C mol-1]
    F_FARADAY = 9.6485341539e4;
    # Absolute zero [Celsius]
    T_ABS = -273.15
    # Charge on the electron [C]
    Q_ELECTRON = -1.60217646263e-19
    # Electrical permittivity of free space [F m-1]
    EPSILON_0 = 8.854187817e-12
    # Magnetic permeability of free space [ 4p10-7 H m-1 (N A-2)]
    MU_0 = math.pi*4.0e-7


c = 1 / math.sqrt(Physics.EPSILON_0 * Physics.MU_0)
print("Speed of light (calulated): %s m/s" %c)
print("Speed of light (table): %s  m/s" %Physics.C_LIGHT)
```

2.3.2 Méthodes de classe

Des méthodes qui ne font pas référence à des attributs d'instance sont des méthodes de classe et elles ne nécessitent pas le paramètre `self` en première position. De plus, elles sont décorées à l'aide du décorateur `@staticmethod` qui doit figurer sur la ligne précédent la définition de la méthode :

```
# ---------------- class OhmsLaw ----------------
class OhmsLaw():
    @staticmethod
    def U(R, I):
        return R * I

    @staticmethod
    def I(U, R):
        return U / R

```

```
    @staticmethod
    def R(U, I):
        return U / I

r = 10
i = 1.5

u = OhmsLaw.U(r, i)
print("Voltage = %s V" %u)
```

À retenir
- Les variables de classe sont partagées par toutes les instances d'une même classe et n'occupent qu'un espace mémoire commun pour toutes les instances. On y accède avec la syntaxe

  ```
  NomClasse.variable
  ```
- Une application type des variables de classe consiste à maintenir le nombre d'instance de la classe en question (par exemple le nombre d'acteurs de cette classe dans un jeu).
- Les méthodes de classe ne peuvent accéder qu'aux variables de classe et ne peuvent donc pas accéder aux variables d'instance. Leur définition suit la syntaxe :

  ```
  class MaClasse(object):

      @staticmethod
      def methode_de_classe(arg1, arg2)
  ```

  et n'utilisent pas le paramètre `self` en première position. Elles sont appelées avec la syntaxe

  ```
  MaClasse.methode_de_classe(arg1=val1, arg2=val2)
  ```

2.4 Jeux d'arcades (Gestion du clavier / conception de jeux)

2.4.1 Objectifs

Dans cette section, tu vas apprendre les éléments suivantes
- Comment réaliser un jeu d'arcades bien connu : Frogger
- Comment gérer les événements du clavier grâce aux *callbacks*
- Gestion des collisions entre les objets du jeu
- Raffraichissement rapide et fluide (25 fps) de l'image
- Comment développer un jeu un peu plus compliqué (environ 100 lignes de code)

2.4.2 Scénario du jeu et cahier des charges

Avant de se lancer dans le codage d'un jeu, il faut définir clairement les règles et les fonctionnalités qu'il devra mettre à disposition. En somme, il faut écrire le scénario du jeu. En génie logiciel et dans la gestion de projet, cela s'appelle "dresser le cahier des charges" du projet. Il s'agit d'un document qui décrit de manière succincte, par des mots-clés et des phrases très courtes, mais dans les détails, toutes les fonctionnalités désirées dans le produit fini.

La plupart du temps, le premier scénario projeté est trop compliqué et il ne sera pas possible de l'implémenter du premier coup. Le programmeur doit donc simplifier le cahier des charges du jeu (logiciel) jusqu'à ce qu'il soit en mesure de l'implémenter pour une version 0.1. Il sera ensuite possible de rajouter des règles, des variantes, des effets et fonctionnalités supplémentaires. Cela nécessite cependant d'écrire le code de façon modulable et bien structurée pour pouvoir rajouter les fonctionnalités ultérieures avec le minimum de modifications. Un développement intelligent qui permet un code flexible et évolutif constitue un véritable défi même pour les meilleurs programmeurs.

Cependant, développer son propre jeu / programme et pouvoir l'exécuter est extrêmement gratifiant, surtout s'il est utile et apprécié par d'autres personnes. Dans notre cas, ce n'est pas tant le produit fini que les compétences et les connaissances que tu vas acquérir pendant le développement (et le débogage ! ! !)

Cahier des charges

À compléter

Le cahier des charges pour ce jeu n'est pas encore en ligne.

2.4.3 Principes de développement

Étape 1

Il y a plusieurs façons de créer un petit jeu tel que Frogger. Voici celle que nous allons adopter :

1. D'abord les mouvements de la grenouille et ceux des véhicules
2. Gestion des collisions
3. Comptage des points et gestion de la fin du jeu

Les voitures sont représentées par des instances de la classe `Car` qui dérive de la classe `Actor`. Dans leur méthode `act()`, on programme la direction dans laquelle elles se déplacent. On utilisera les sprites `car0.gif` jusqu'à `car19.gif` qui se trouvent dans l'archive TigerJython. Il est possible d'utiliser tes propres images, mais leur taille ne doit pas dépasser 70 pixels de hauteur et 200 pixels de largeur.

Pour les jeux d'arcades, on utilise généralement une grille de jeu dont la taille des carrés est 1x1 pixel, ce qui fait correspondre chaque carré de la grille à un pixel de l'écran. Nous utiliserons une grille de taille 800×600 pour commencer, ce qui paraîtra un peu petit sur des écrans haute définition. Dans la fonction `initCars()`, tu peux générer les 20 véhicules en déterminant leur lieu de création et leur sens de déplacement.

Le déplacement des véhicules avec la méthode `act()` est simple : dès qu'ils sortent de la grille de jeu d'un côté, tu peux les faire réapparaitre de l'autre côté. Il faut noter à cet effet que l'acteur existe encore dans le moteur du jeu même s'il se trouve en dehors de la grille. Il est donc possible de le faire apparaître de l'autre côté de la grille, mais en dehors de la zone visible.

Code de base

```
from gamegrid import *

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
```

```
17          car = Car("sprites/car" + str(i) + ".gif")
18          if i < 5:
19              addActor(car, Location(350 * i, 100), 0)
20          if i >= 5 and i < 10:
21              addActor(car, Location(350 * (i - 5), 220), 180)
22          if i >= 10 and i < 15:
23              addActor(car, Location(350 * (i - 10), 350), 0)
24          if i >= 15:
25              addActor(car, Location(350 * (i - 15), 470), 180)
26
27
28  makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False)
29  setSimulationPeriod(50)
30  initCars()
31  show()
32  doRun()
```

Étape 2 (Bouger la grenouille avec le clavier)

Nous allons maintenant faire intégrer la grenouille dans le jeu. Le joueur la bougera grâce aux touches haut, bas, gauche, droite du clavier. Ensuite, il faut faire le suivant :

– Définir une classe `Frog` qui dérive de la classe `Actor`. Aucune méthode n'est nécessaire hormis le construteur puisque ce seront des événements clavier qui vont la faire bouger.

– Définir le gestionnaire d'événements `keyCallback()` qui sera signalé au moteur de jeu comme gestionnaire d'événements du clavier avec

```
makeGameGrid(..., keyRepeated=keyCallback)
```

On associe le gestionnaire `keyCallback` au paramètre `keyRepeated` pour qu'il ne soit pas uniquement appelé de manière unique lorsqu'on appuie courtement sur une touche, mais qu'il le soit de manière rapide et répétée lorsque la touche est maintenue enfoncée.

Astuce : Attention à **ne pas mettre de parenthèses** lorsque le gestionnaire est passé en paramètre à la fonction `makeGameGrid`. Il faut bien passer la fonction en tant qu'objet et non sa valeur de retour avec `makeGameGrid(...)`

– Dans le gestionnaire `keyCallback`, le paramètre `keyCode` sera renseigné avec le code (nombre entier) de la touche pressée. Il suffit de déplacer la grenouille de 5 pixels dans la direction indiquée sur la base de `keyCode` ou de ne rien faire si une autre touche a été pressée.
Pour augmenter la lisibilité du programme, on définit les constantes constantes suivantes en début de code :

```
# ---------------- Constantes clavier --------
K_LEFT      = 37
K_UP        = 38
K_RIGHT     = 39
K_DOWN      = 40
```

Astuce : Si tu ne connais pas le `keyCode` d'une touche du clavier, il suffit d'essayer en exécutant ce programme qui fait appel à un gestionnaire d'événements qui affiche le code reçu avec un `print` :

```
from gamegrid import *

def pressCallback(e):
    print "Press: ", e.getKeyCode()

def releaseCallback(e):
```

```
    print "Release: ", e.getKeyCode()

makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
             keyPressed = pressCallback,
             keyReleased = releaseCallback)
show()
```

Code

```
from gamegrid import *

# ---------------- Constantes clavier --------
K_LEFT      = 37
K_UP        = 38
K_RIGHT     = 39
K_DOWN      = 40

# ---------------- classe Frog ---------------
class Frog(Actor):
  def __init__(self):
      Actor.__init__(self, "sprites/frog.gif")

# ---------------- classe Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        if i < 5:
            addActor(car, Location(350 * i, 100), 0)
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)

def keyCallback(keyCode):
    if keyCode == K_LEFT:
        frog.setX(frog.getX() - 5)
    elif keyCode == K_UP:
        frog.setY(frog.getY() - 5)
    elif keyCode == K_RIGHT:
        frog.setX(frog.getX() + 5)
    elif keyCode == K_DOWN:
        frog.setY(frog.getY() + 5)
```

```
47
48
49  makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
50               keyRepeated = keyCallback)
51  setSimulationPeriod(50);
52  frog = Frog()
53  addActor(frog, Location(400, 560), 90)
54  initCars()
55  show()
56  doRun()
```

2.4.4 Détection de collisions

Il est très facile d'installer une détection des collisions avec le module *JGameGrid* car il s'occupe de faire les calculs géométriques pour nous. Il suffit, lors de la création d'une instance `car` de la classe `Car`, d'appeler la méthode `addCollisionActor` de la grenouille :

```
frog.addCollisionActor(car)
```

Cet appel va indiquer à l'instance `frog` de la classe `Frog` que lors de chaque collision avec l'acteur `car`, elle doit générer un événement spécial qui va se charger d'invoquer la méthode `Actor.collide()` de l'instance `frog`. C'est dans cette méthode qu'on peut ensuite décider de la manière de réagir à la collision, en l'occurrence faire sauter la grenouille à sa position de départ.

```
1   from gamegrid import *
2
3   # ---------------- Constantes clavier --------
4   K_LEFT      = 37
5   K_UP        = 38
6   K_RIGHT     = 39
7   K_DOWN      = 40
8
9   # ---------------- classe Frog ---------------
10  class Frog(Actor):
11      def __init__(self):
12          Actor.__init__(self, "sprites/frog.gif")
13
14      def collide(self, actor1, actor2):
15          self.setLocation(Location(400, 560))
16          return 0
17
18  # ---------------- classe Car ----------------
19  class Car(Actor):
20      def __init__(self, path):
21          Actor.__init__(self, path)
22
23      def act(self):
24          self.move()
25          if self.getX() < -100:
26              self.setX(1650)
27          if self.getX() > 1650:
28              self.setX(-100)
29
30  def initCars():
31      for i in range(20):
32          car = Car("sprites/car" + str(i) + ".gif")
33          frog.addCollisionActor(car)
```

```
34          if i < 5:
35              addActor(car, Location(350 * i, 100), 0)
36          if i >= 5 and i < 10:
37              addActor(car, Location(350 * (i - 5), 220), 180)
38          if i >= 10 and i < 15:
39              addActor(car, Location(350 * (i - 10), 350), 0)
40          if i >= 15:
41              addActor(car, Location(350 * (i - 15), 470), 180)
42
43  def keyCallback(keyCode):
44      if keyCode == K_LEFT:
45          frog.setX(frog.getX() - 5)
46      elif keyCode == K_UP:
47          frog.setY(frog.getY() - 5)
48      elif keyCode == K_RIGHT:
49          frog.setX(frog.getX() + 5)
50      elif keyCode == K_DOWN:
51          frog.setY(frog.getY() + 5)
52
53
54  makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
55               keyRepeated = keyCallback)
56  setSimulationPeriod(50);
57  frog = Frog()
58  addActor(frog, Location(400, 560), 90)
59  initCars()
60  show()
61  doRun()
```

La méthode `collide()`

La méthode `collide()` est définie à la base dans la classe `Actor` sans comportement particulier et doit être surchargée (*overriden*) dans la classe `Frog` pour implémenter le comportement désiré.

Par défaut, la détection se fait par calcul géométrique avec les coordonnées des sommets des rectangles entourants les sprites, comme représenté sur la figure ci-dessous :

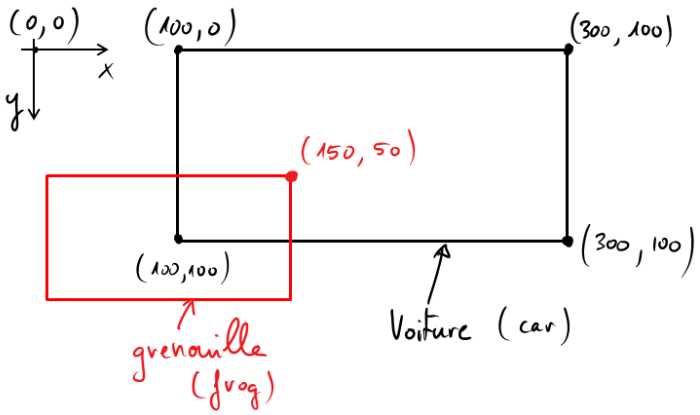


FIGURE 2.4 – Illustration de la détection de collision avec la méthode des rectangles

Il est possible de personnaliser ce comportement en précisant la région de l'espace qui sera prise en compte pour la détection de la superposition. Voici les méthodes utiles à cet effet permettant de spécifier la forme, la taille et les

coordonnées de la zone du sprite sensible à la collision :
- `setCollisionCircle(centerPoint, radius)` : Cercle de centre `centerPoint` et de rayon `radius` (en pixels)
- `setCollisionImage()` : Nicht-transparente Bildpixels (nur mit einem Partner der Kreis, Linie oder Punkt als Kollisionsarea hat)
- `setCollisionLine(startPoint, endPoint)` : Segment de droite dont les extrémités sont les points `startPoint` et `endPoint`
- `setCollisionRectangle(center, width, height)` : Rectangle de centre `center`, de largeur `width` et de hauteur `height`
- `setCollisionSpot(spotPoint)` : Un point particulier de l'image

Remarque

Pour toutes les méthodes décrites ci-dessus, le système de coordonnées utilisé est relatif au sprite. Son origine se trouve au centre du rectangle délimitant le sprite et l'axe Oy est dirigé vers le bas :

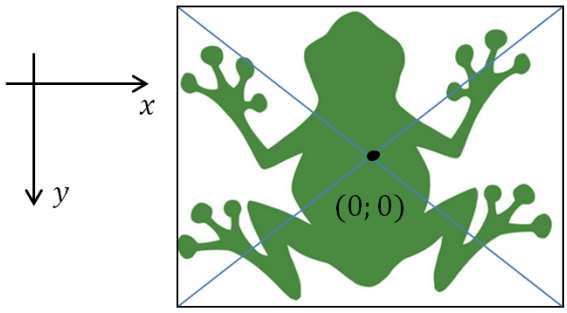
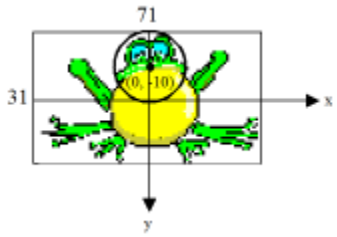


FIGURE 2.5 – Système d'axe relatif au sprite

Le sprite de la grenouille a une taille de 71×41 pixels. On peut changer la zone de la grenouille sensible aux collisions dans le construteur de la classe `Frog` :

```
self.setCollisionCircle(Point(0, -10), 5)
```

de sorte qu'une collision sera générée lorsqu'une voiture roule sur le cercle de centre $(0; -15)$ et de rayon 5 qui entoure sa tête :



2.4.5 Moteur de jeu

Maintenant que nous avons développé tout ce qui tourne autour des mouvements des acteurs de notre jeu, il nous faut implémenter la logique du jeu, à savoir le comptage des points et les conditions de fin de jeu.

Pour ce faire, on pourrait utiliser des variables globales pour stocker le nombre de fois que la grenouille a traversé avec succès toute la route et le nombre de fois que la grenouille a été écrasée.

Si la grenouille a trois vies, on peut stoper le jeu et afficher un message de *Game Over* lorsqu'elle s'est faite écraser trois fois.

Mais comme vous le savez, l'utilisation de variables globales n'est pas indiquée dans le 99% des cas et le nôtre ne fait pas exception. Le mieux serait de créer une classe `FroggerGame` qui va stocker ces différents paramètres en tant que variables d'instances.

2.4.6 Extensions / Exercices

1. Créer une nouvelle classe `FroggerGame` pour pour modéliser le jeu dans une classe qui constituera le programme principal. Il ne doit y avoir aucune variable dans l'espace de noms global.

Consignes
- La grenouille a trois vies
- À chaque fois que la grenouille traverse la route, augmenter un compteur `nb_succes` défini judicieusement
- À chaque fois que la grenouille se fait écraser, diminuer le nombre de vies de 1
- Afficher dans le titre de la fenêtre le nombre de succès et le nombre de vies restantes

---

Indications
- On peut afficher la chaine `texte` dans la barre de titre de la fenêtre avec

  ```
  setTitle(texte)
  ```

- On peut mettre le jeu en pause avec

  ```
  doPause()
  ```

- On sait que la fenêtre principale a été fermée lorsque `isDisposed() == True`
- Pour afficher le *Game Over*, on peut créer un acteur `gameOver` avec le sprite `"sprites/gameover.gif"` ou avec un sprite personnalisé dont le fond est transparent.

Code de base

---

2. Jouer le son `"wav/boing.wav"` lorsque la grenouille se fait écraser et le son `"wav/notify.wav"` lorsqu'elle parvient avec succès à traverser la route.
3. Faire un comptage de points qui sera affiché dans la barre de titre de la fenêtre.
   - Lorsque la grenouille traverse avec succès, le joueur marque 5 points.
   - Lorsque la grenouille se fait écraser, le joueur perd 5 points
4. Inclure un temps limite pour chaque traversée de la route. Si le temps est dépassé, supprimer 10 points et remettre la grenouille au point de départ.
5. Fais en sorte que les voitures ne roulent pas à la même vitesse sur toutes les voies.
6. Au lieu d'avoir un décallage régulier entre les voitures, introduis une distance aléatoire comprise entre 20 et 100 pixels.
7. Sois créatif et ajoute tes propres fonctionnalités au jeu.

CHAPITRE 3

Annexes

3.1 Installation de TigerJython sous Linux

Pour installer *TigerJython* (http://jython.tobiaskohn.ch/) sous Linux, entrer les commandes suivantes dans un terminal

```
# création du dossier pour tigerjython
mkdir tigerjython && cd tigerjython

# installation de la JRE (Java Runtime Environment)
sudo apt-get install default-jre wget

# aller dans le dossier de tigerjython et télécharger TigerJython
cd tigerjython && wget http://jython.tobiaskohn.ch/tigerjython.jar
```

3.1.1 Exécuter tigerjython

Pour exécuter TigerJython, il suffit d'entrer la commande suivante :

```
# exécution de l'archive Java de TigerJython
java -jar tigerjython.jar
```

3.2 Le langage de modélisation UML

3.2.1 Références

– http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/
– http://creately.com/blog/diagrams/class-diagram-relationships/

3.2.2 Tutoriels et cours

– http://fr.openclassrooms.com/mooc/debutez-analyse-logicielle-avec-uml