Programmation orientée objet :

Paul-Antoine BISGAMBIGLIA
Faculty of Science
UMR-CNRS-6134
University of Corsica – 20250 Corte
Bisgambiglia_pa@univ-corse.fr

Mots-clés : POO, python

Bonjour à tous,

Voici le déroulé du cours de POO

24 octobre :

1. Point cours
2. Exercice de positionnement
3. Exercice fil rouge (en lien avec le cours de dev. Web en python)

14 novembre :

1. Point cours
2. Exercice fil rouge (en lien avec le cours de dev. Web en python)

18 novembre :

3. Exercice fil rouge (en lien avec le cours de dev. Web en python)
4. Evaluation, présentation de vos codes.

Rappels

1. Concepts de base de la Programmation Orientée Objet (POO) en Python

Voici une liste des concepts de base de la POO :

1. **Classe :**
   Une classe est une structure qui permet de créer des objets. Elle définit les attributs (variables) et les méthodes (fonctions) que ces objets auront.

```python
class Animal:
    def __init__(self, nom):
        self.nom = nom
```

2. **Objet :**
   Un objet est une instance d'une classe. Chaque objet possède son propre ensemble de données défini par la classe.
   ```python
   chien = Animal("Rex")
   ```

3. **Encapsulation :**
   L'encapsulation consiste à restreindre l'accès direct aux attributs d'un objet et à utiliser des méthodes pour y accéder ou les modifier. Python utilise des conventions pour cela, comme les variables privées (précédées de `_` ou `__`).
   ```python
   class Animal:
       def __init__(self, nom):
           self.__nom = nom  # attribut privé

       def get_nom(self):
           return self.__nom

       def set_nom(self, nom):
           self.__nom = nom
   ```

4. **Héritage :**
   L'héritage permet de créer une nouvelle classe à partir d'une classe existante. La nouvelle classe hérite des attributs et méthodes de la classe parent.
   ```python
   class Chien(Animal):
       def __init__(self, nom, race):
           super().__init__(nom)  # Appel du constructeur parent
           self.race = race
   ```

5. **Polymorphisme :**
   Le polymorphisme permet d'utiliser une méthode de différentes façons, en fonction du type de l'objet. Dans Python, cela peut se faire grâce à l'héritage, aux méthodes redéfinies ou à l'utilisation des classes abstraites.
   ```python
   class Animal:
       def parler(self):
   ```

```
        pass

class Chien(Animal):
    def parler(self):
        return "Woof!"

class Chat(Animal):
    def parler(self):
        return "Meow!"

animaux = [Chien("Rex"), Chat("Felix")]

for animal in animaux:
    print(animal.parler())
```

6. **Abstraction :**
   L'abstraction consiste à ne montrer que les fonctionnalités essentielles d'un objet tout en cachant les détails de son implémentation. On peut créer des classes abstraites en Python avec le module `abc`.

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def parler(self):
        pass
```

7. **Méthodes et Attributs de Classe (vs d'Instance) :**
   Les méthodes ou attributs de classe sont partagés entre toutes les instances d'une classe, tandis que les méthodes et attributs d'instance sont spécifiques à chaque objet.

```python
class Animal:
    nombre_animaux = 0  # Attribut de classe

    def __init__(self, nom):
        self.nom = nom  # Attribut d'instance
        Animal.nombre_animaux += 1  # Modifier l'attribut de classe
```

8. **Propriétés (getters et setters avec @property) :**
   En Python, vous pouvez utiliser les décorateurs `@property` pour définir des getters et des setters de manière concise.

```python
class Animal:
    def __init__(self, nom):
        self._nom = nom

    @property
    def nom(self):
        return self._nom

    @nom.setter
    def nom(self, nouveau_nom):
        self._nom = nouveau_nom
```

2. Encapsulation

Source des codes : https://www.pythoncheatsheet.org/cheatsheet/oop-basics

```python
# Define a class named MyClass
class MyClass:

    # Constructor method that initializes the class object
    def __init__(self):

        # Define a protected variable with an initial value of 10
        # The variable name starts with a single underscore, which indicates protected access
        self._protected_var = 10

        # Define a private variable with an initial value of 20
        # The variable name starts with two underscores, which indicates private access
        self.__private_var = 20

# Create an object of MyClass class
obj = MyClass()

# Access the protected variable using the object name and print its value
```

```
# The protected variable can be accessed outside the class but
# it is intended to be used within the class or its subclasses
print(obj._protected_var)   # output: 10

# Try to access the private variable using the object name and print its value
# The private variable cannot be accessed outside the class, even by its subclasses
# This will raise an AttributeError because the variable is not accessible outside the class
print(obj.__private_var)    # AttributeError: 'MyClass' object has no attribute '__private_var'
```

3. Héritage

```
# Define a class named Animal
class Animal:

    # Constructor method that initializes the class object with a name attribute
    def __init__(self, name):
        self.name = name

    # Method that is defined in the Animal class but does not have a body
    # This method will be overridden in the subclasses of Animal
    def speak(self):
        print("")

# Define a subclass named Dog that inherits from the Animal class
class Dog(Animal):

    # Override the speak method of the Animal class
    def speak(self):
        print("Woof!")

# Define a subclass named Cat that inherits from the Animal class
class Cat(Animal):

    # Override the speak method of the Animal class
    def speak(self):
        print("Meow!")

# Create a Dog object with a name attribute "Rover"
```

```
dog = Dog("Rover")

# Create a Cat object with a name attribute "Whiskers"
cat = Cat("Whiskers")

# Call the speak method of the Dog class and print the output
# The speak method of the Dog class overrides the speak method of the Animal class
# Therefore, when we call the speak method of the Dog object, it will print "Woof!"
dog.speak()   # output: Woof!

# Call the speak method of the Cat class and print the output
# The speak method of the Cat class overrides the speak method of the Animal class
# Therefore, when we call the speak method of the Cat object, it will print "Meow!"
cat.speak()   # output: Meow!
```

4. Polymorphisme

```
class Rectangle(Shape):
    # The Rectangle class is defined with an __init__ method that initializes
    # width and height instance variables.
    # It also defines an area method that calculates and returns
    # the area of a rectangle using the width and height instance variables.
    def __init__(self, width, height):
        self.width = width  # Initialize width instance variable
        self.height = height  # Initialize height instance variable

    def area(self):
        return self.width * self.height  # Return area of rectangle


# The Circle class is defined with an __init__ method
# that initializes a radius instance variable.
# It also defines an area method that calculates and
# returns the area of a circle using the radius instance variable.
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius  # Initialize radius instance variable
```

```
    def area(self):
        return 3.14 * self.radius ** 2  # Return area of circle using pi * r^2

# The shapes list is created with one Rectangle object and one Circle object. The for
# loop iterates over each object in the list and calls the area method of each object
# The output will be the area of the rectangle (20) and the area of the circle (153.86).
shapes = [Rectangle(4, 5), Circle(7)]  # Create a list of Shape objects
for shape in shapes:
    print(shape.area())  # Output the area of each Shape object
```

5. Classe abstraite

```
# Import the abc module to define abstract classes and methods
from abc import ABC, abstractmethod

# Define an abstract class called Shape that has an abstract method called area
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

# Define a Rectangle class that inherits from Shape
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    # Implement the area method for Rectangles
    def area(self):
        return self.width * self.height

# Define a Circle class that also inherits from Shape
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    # Implement the area method for Circles
    def area(self):
```

```
        return 3.14 * self.radius ** 2


# Create a list of shapes that includes both Rectangles and Circles
shapes = [Rectangle(4, 5), Circle(7)]


# Loop through each shape in the list and print its area
for shape in shapes:
    print(shape.area())
```

6. Exercice zoo

Vous allez créer un mini-système de gestion de zoo. Ce système permet de gérer différents types d'animaux (par exemple, chiens, chats, lions, éléphants). Chaque animal peut "parler" en fonction de son type (polymorphisme), et vous allez également gérer le nombre total d'animaux dans le zoo.

Objectifs :

1. Créez une classe abstraite **Animal** avec les attributs `nom` et `age` et une méthode abstraite `parler()`.
2. Implémentez plusieurs classes dérivées d'**Animal** telles que **Chien**, **Chat**, et **Lion** qui implémentent la méthode `parler()`.
3. Implémentez un compteur d'animaux (utilisez des attributs de classe pour cela).
4. Utilisez l'encapsulation pour protéger l'accès aux attributs des animaux.
5. Implémentez un getter et un setter pour l'âge de l'animal en utilisant `@property`.
6. Créez une classe **Zoo** qui permet de stocker plusieurs animaux et d'afficher des informations sur eux (par exemple, leur nom, leur âge, et leur son).

7. Exercice RPG

Si dessous un code python 2.7, sur la base de ce code et de l'énoncé de l'atelier bilan POO de L3 (sous le code), vous devez développer un code python qui simule un RPG avec 3 types de parties possibles :

1. Le mode quête, le joueur affronte des ennemies
2. Le mode versus, deux joueurs s'affrontent
3. Le mode plateau, inspiré de l'énoncé de l'atelier bilan POO.

Ce code devra être intégré dans votre code serveur du cours de développement web et une interface web devra permettre de jouer.

```
### import
import math
import random
```

```
from datetime import date
class Stat:
    """ stat of the player """
    def __init__(self,dictArgs):
        self.strength  = dictArgs['strength']
        self.magic  = dictArgs['magic']
        self.agility  = dictArgs['agility']
        self.speed  = dictArgs['speed']
        self.charisma  = dictArgs['charisma']
        self.chance  = dictArgs['chance']
        self.endurance = random.randint(self.strength+self.agility,2 * (self.strength + self.agility))
        self.life_point = random.randint(self.endurance,2 * self.endurance)
        self.attack = self.strength + self.magic + self.agility
        self.defense = self.agility + self.speed + self.endurance

    @property
    def strength(self):
        print "strength: "
        return self._strength

    @strength.setter
    def strength(self, v):
        print "strength upgrade"
        self._strength  =  v

    def __str__(self):
        return str(self.__dict__)

class Classe:
    """ class type """
    def __init__(self,name,stat):
        self._name = name
        self._stat = stat

    def __str__(self):
        return str(self._name)

class Race:
    """ race type """
    def __init__(self,name,stat):
```

```
        self._name = name
        self._stat = stat

    def __str__(self):
        return str(self._name)

class Avatar:
    """ general class """
    id = 0
    def __init__(self,targs):
        self._nom = targs['name']
        self._race = targs['race']
        self._classe = targs['classe']
        self._bag = targs['bag']
        self._equipment = targs['equipment']
        self._element = targs['element']
        self._lvl = 1
        self._stat = Stat({'strength':1, 'magic':1,'agility':1,'speed':1,'charisma':0,'chance':0})
        Avatar.id += 1
        self._id = Avatar.id
        self.sumStat()
        self._life = self._stat.life_point
        self._statistics = {"fight":0,"win":0,"maxDomage":0}

    def getBag(self):
        return self._bag._lItems

    def initiative(self):
        min = self._stat.speed
        max = self._stat.agility + self._stat.chance + self._stat.speed
        return random.randint(min,max)

    def damages(self):
        critique = random.randint(0,self._stat.chance)
        min = 0
        max = self._stat.attack
        if (critique > self._stat.chance / 2):
            print ("full damages")
            maxDam = random.randint(max, 2 * max)
        else:
```

```
            maxDam = random.randint(min,max)
        print self._nom + " done " + str(maxDam)
        return maxDam

    def defense(self,v):
        min = self._stat.agility
        max = self._stat.agility + self._stat.chance + self._stat.speed
        duck = random.randint(min,max)
        damage = 0
        if (duck == max):
            print ("the shot is dodged")
        elif (duck > max / 2):
            print ("partial dodge")
            damage /= 2
        else:
            damage = v
        damage -= self._stat.defense
        if (damage < 0):
            damage = 0
        if (damage > self._life):
            self._life = 0
            print ("You are dead")
        else:
            self._life -= damage
        print "life point: ",self._life," / ",self._stat.life_point

    def __str__(self):
        show = str(self._nom)
        return show

    def sumStat(self):
        equiment = 0
        for i in self._stat.__dict__:
            for j in self._equipment:
                equiment += j._stat.__dict__[i]
            self._stat.__dict__[i] = self._race._stat.__dict__[i] + self._classe._stat.__dict__[i] + equiment

class Mobs(Avatar):
    def __init__(self,targs):
        Avatar.__init__(self,targs)
```

```
        self._type = targs["type"]

    def __str__(self):
        output = "Mobs " + self._type + " " + self._nom
        return output

class Hero(Avatar):
    def __init__(self,targs):
        Avatar.__init__(self,targs)
        self._xp = 1
        self._profession = targs['profession']
        self._lvl = self.lvl()

    def lvl(self):
        lvl = math.floor(self._xp/100)
        if (lvl < 1):
            lvl = 1
        if (lvl > self._lvl):
            print "### new level ###"
            self.newLvl()
        return lvl

    def newLvl(self):
        for i in self._stat.__dict__:
            self._stat.__dict__[i] += 5
        self._life = self._stat.life_point
        print "### stats upgrade ###"

    def setXP(self,xp):
        self._xp += xp
        self._lvl = self.lvl()

    def __str__(self):
        output = "joueur " + self._nom + " de niveau " + str(self._lvl) + " classe " + str(self._classe) + " race " +
str(self._race)
        return output

    def save(self):
        fileName = str(date.today())+"_"+str(Hero.id)+"_"+str(self._nom)+".txt"
        f = open(fileName,"w+")
```

```
        f.write(self._nom + "\n")
        f.write(self._race._name + "\n")
        f.write(self._classe._name + "\n")
        f.write("lvl: " + str(self._lvl) + "\n")
        f.write("xp: " + str(self._xp) + "\n")
        for i in self._stat.__dict__:
            output = str(i) + " " + str(self._stat.__dict__[i])
            f.write(output + "\n")
        for i in self._equipment:
            f.write(str(i) + "\n")
        for i in self.getBag():
            f.write(str(i) + "\n")
        f.close()

    def saveXML(self):
        fileName = str(date.today())+"_"+str(Hero.id)+"_"+str(self._nom)+".xml"
        f = open(fileName,"w+")
        xml = "<?xml version='1.0' encoding='UTF-8'?>"
        xml += "<avatar id='"+ str(Hero.id) +"'>"
        xml += "<name>" + self._nom + "</name>"
        xml += "<race>" + self._race._name + "</race>"
        xml += "<level>" + self._classe._name + "</level>"
        xml += "<xp>" + str(self._lvl) + "</xp>"
        xml += "<name>" + str(self._xp) + "</name>"
        xml += "<stats>"
        for i in self._stat.__dict__:
            xml += "<" + str(i) + ">" + str(self._stat.__dict__[i]) + "</" + str(i) + ">"
        xml += "</stats>"
        xml += "<equipments>"
        it = 1
        for i in self._equipment:
            xml += "<item_" + str(it) + ">" + i._name + "</item_" + str(it) + ">"
            it += 1
        xml += "</equipments>"
        xml += "<bag>"
        it = 1
        for i in self.getBag():
            xml += "<item_" + str(it) + ">" + i._name + "</item_" + str(it) + ">"
            it += 1
        xml += "</bag>"
```

```
        xml += "</avatar>"
        f.write(xml)
        f.close()

    @staticmethod
    def load():
        pass

class Item:
    """ object class """
    nbr = 0
    def __init__(self,targs,stat):
        self._name = targs['name']
        self._type = targs['type']
        self._space = targs['space']
        self._stat = stat
        Item.nbr += 1

    def __str__(self):
        return str(self._name)

class Equipment(Item):
    def __init__(self,targs,stat):
        Item.__init__(self,targs,stat)
        self._lClasses = targs['classList']
        self._place = targs['place']

class Bag:
    """ Bag class to save Items """
    def __init__(self,args):
        self._sizeMax = args['sizeMax']
        self._lItems = args['items']
        self._size = len(self._lItems)

    def addItem(self,i):
        if (self._size < self._sizeMax):
            self._lItems.append(i)
            self._size += 1
        else:
            return False
```

```
    def delItem(self,i):
        self._lItems.pop(i)
        self._size -= 1

    def __str__(self):
        output = ""
        for i in self._lItems:
            output += str(i)
        return output

class Quest:
    """ class for manage quest """
    def __init__(self,targs):
        self._lAvatar = targs['lAvatar']
        self._lvl = targs['lvl']
        self._itemGift = targs['gift']

    def run(self,hero):
        round = 1
        output = ""
        if (len(self._lAvatar) == 1):
            output += "### PVP MODE ####"
            print "### PVP MODE ####"
            player = self._lAvatar[0]
            output += "\n" + player._nom + " VS " + hero._nom
            print player._nom + " VS " + hero._nom
            while (player._life > 0 ) and (hero._life > 0 ):
                output += "\n" + "Round " + str(round)
                print "# Round " + str(round) + " # "
                print "# PV de " + hero._nom + " " + str(hero._life)
                output += "\n" + "# PV de " + hero._nom + " " + str(hero._life)
                print "# PV de " + player._nom + " " + str(player._life)
                output += "\n" + "# PV de " + player._nom + " " + str(player._life)
                if (player.initiative() > hero.initiative):
                    output += "\n" + player._nom + " begin"
                    print player._nom + " begin"
                    hero.defense(player.damages())
                    if (hero._life <= 0):
                        output += "\n" + player._nom + " win"
```

```
                print player._nom + " win"
            else:
                player.defense(hero.damages())
        else:
            output += "\n" + hero._nom + " begin"
            print hero._nom + " begin"
            player.defense(hero.damages())
            if (player._life <= 0):
                output += "\n" + hero._nom + " win"
                print hero._nom + " win"
            else:
                hero.defense(player.damages())
        round += 1
    if (hero._life <= 0):
        output += "\n" + player._nom + " win"
        print player._nom + " win"
        player.setXP(10*self._lvl)
        player._bag.addItem(self._itemGift)
    else:
        output += "\n" + hero._nom + " win"
        print hero._nom + " win"
        hero.setXP(10*self._lvl)
        hero._bag.addItem(self._itemGift)
else:
    output += "\n" + "### Quest MODE ####"
    print "### Quest MODE ####"
    for player in self._lAvatar:
        output += "\n" + player._nom + " VS " + hero._nom
        print player._nom + " VS " + hero._nom
        while (player._life > 0 ) and (hero._life > 0 ):
            output += "\n" + "Round " + str(round)
            print "Round " + str(round)
            print "# Round " + str(round) + " # "
            print "# PV de " + hero._nom + " " + str(hero._life)
            output += "\n" + "# PV de " + hero._nom + " " + str(hero._life)
            print "# PV de " + player._nom + " " + str(player._life)
            output += "\n" + "# PV de " + player._nom + " " + str(player._life)
            if (player.initiative() > hero.initiative):
                output += "\n" + player._nom + " begin"
                print player._nom + " begin"
```

```
                tmpDegats = player.damages()
                hero.defense(tmpDegats)
                output += "\n" + player._nom + " degats " + str(tmpDegats)
                print player._nom + " degats " + str(tmpDegats)
                if (hero._life <= 0):
                    output += "\n" + player._nom + " win"
                    print player._nom + " win"
                else:
                    tmpDegats = hero.damages()
                    player.defense(tmpDegats)
                    output += "\n" + hero._nom + " degats " + str(tmpDegats)
                    print hero._nom + " degats " + str(tmpDegats)
            else:
                output += "\n" + hero._nom + " begin"
                print hero._nom + " begin"
                tmpDegats = hero.damages()
                player.defense(tmpDegats)
                output += "\n" + hero._nom + " degats " + str(tmpDegats)
                print hero._nom + " degats " + str(tmpDegats)
                if (player._life <= 0):
                    output += "\n" + hero._nom + " win"
                    print hero._nom + " win"
                else:
                    tmpDegats = player.damages()
                    hero.defense(tmpDegats)
                    output += "\n" + player._nom + " degats " + str(tmpDegats)
                    print player._nom + " degats " + str(tmpDegats)
            round += 1
        if (hero._life <= 0):
            output += "\n" + "You loose"
            print "You loose"
        else:
            output += "\n" + hero._nom + " win"
            print hero._nom + " win"
            hero.setXP(10 * len(self._lAvatar) * self._lvl)
            hero._bag.addItem(self._itemGift)
        return output

    def __str__(self):
        return self._itemGift
```

```
def main():
    ### RACE
    statElfe = Stat({'strength':5, 'magic':10,'agility':10,'speed':5,'charisma':5,'chance':5})
    elfe = Race('Elfe',statElfe)
    statHuman = Stat({'strength':10, 'magic':10,'agility':5,'speed':5,'charisma':5,'chance':5})
    human = Race('Human',statHuman)
    statDwarf = Stat({'strength':10, 'magic':0,'agility':10,'speed':5,'charisma':5,'chance':10})
    dwarf = Race('Dwarf',statDwarf)
    statOrc = Stat({'strength':15, 'magic':0,'agility':5,'speed':10,'charisma':5,'chance':5})
    orc = Race('Orc',statOrc)
    ### CLASS
    statWizard = Stat({'strength':0, 'magic':10,'agility':0,'speed':0,'charisma':10,'chance':10})
    wizard = Race('Wizard',statWizard)
    statWarrior = Stat({'strength':10, 'magic':0,'agility':5,'speed':5,'charisma':5,'chance':5})
    warrior = Race('Warrior',statWarrior)
    ### ITEMS
    statSword = Stat({'strength':5, 'magic':0,'agility':5,'speed':5,'charisma':0,'chance':5})
    sword = Equipment({'classList':'warrior','place':'hand','name':'dragon
sword','type':'sword','space':2,},statSword)
    statBaton = Stat({'strength':0, 'magic':10,'agility':0,'speed':5,'charisma':0,'chance':5})
    baton = Equipment({'classList':'wizard','place':'hand','name':'wizard baton','type':'baton','space':2,},statBaton)
    statPotion = Stat({'strength':0, 'magic':0,'agility':0,'speed':0,'charisma':0,'chance':0})
    Potion = Item({'name':'life potion','type':'potion','space':2,},statPotion)
    ### BAG
    myBag = Bag({"sizeMax":20,"items":[Potion,Potion]})
    ### MOBS
    mechant1 = Mobs({'name':'orc
1','race':orc,'classe':warrior,'bag':myBag,'equipment':[sword],'element':'Fire','type':'soldier'})
    mechant2 = Mobs({'name':'orc
2','race':orc,'classe':warrior,'bag':myBag,'equipment':[sword],'element':'Fire','type':'soldier'})
    hero1 =
Hero({'name':'Jean','race':elfe,'classe':wizard,'bag':myBag,'equipment':[baton],'element':'Fire','profession':'chom
eur'})
    hero2 =
Hero({'name':'Pierre','race':human,'classe':warrior,'bag':myBag,'equipment':[sword],'element':'Fire','profession':'
chomeur'})
    hero1.save()
    hero1.saveXML()
    ### QUEST
```

```
    firstQuest = Quest({'lAvatar':[mechant1,mechant2],'lvl':2,'gift':sword})
    #firstQuest = Quest({'lAvatar':[hero2],'lvl':2,'gift':sword})
    #firstQuest.run(hero1)


if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Diplôme : **Licence SPI 3ème année**	2023-2024

UE : Ateliers de programmation

Programmation Orientée Objet

Atelier 3 : Bilan

- classes attributs méthodes
- attributs et méthodes statiques
- hiérarchie d'héritage
- classes abstraites, polymorphisme

Enseignants : Paul-Antoine BISGAMBIGLIA, Marie-Laure NIVET, Evelyne VITTORI

Jeu d'aventures

On souhaite développer un simulateur très simplifié de jeu d'aventures.
Le jeu comporte des personnages qui se déplacent dans les cases d'un tableau à une dimension. Chaque personnage appartient à un joueur et lui procure des points ou des pénalités (diminution de points) lors de ses déplacements.
Le jeu comporte deux types de personnages, les taurens et les humains qui se déplacent à des rythmes différents.
Le lancement du jeu consiste à déplacer successivement chacun des personnages. Ce déplacement est répété autant de fois que le jeu comporte d'étapes.
Le jeu comporte 50 cases numérotées de 0 à 49. Lors du lancement du jeu, le tableau des cases est initialisé. Un gain (nombre de points gagnés) est ainsi attribué à chaque case et des obstacles sont placés sur certaines cases. Un obstacle est caractérisé par une pénalité (nombre de points à soustraire).
Lorsque l'on demande à un personnage de se déplacer, il détermine la case sur laquelle il souhaite aller. Si la case est vide, le personnage s'y place et cela procure une augmentation de points à son joueur propriétaire (égale au montant du gain associé à la case). Si la case comporte un obstacle, le personnage ne se déplace pas et cela entraine une diminution de points à son joueur propriétaire (égale au montant de la pénalité associée à l'obstacle). Enfin, si la case souhaitée est déjà occupée par un autre personnage, le personnage ne se déplace pas et cela entraine une diminution de points à son joueur propriétaire (égale au montant du gain associé à la case).

A la fin du jeu, la liste des joueurs est affichée avec leur nombre de points respectifs et le (ou les) gagnants sont identifiés (joueurs ayant obtenu le nombre maximum de points).

Le logiciel doit permettre de créer un jeu, de créer des joueurs, de créer des personnages appartenant à des joueurs, d'inscrire des joueurs au jeu et de lancer le jeu.

Pour chacune des questions, vous prendrez soin d'écrire un programme de test de chaque classe sans attendre la question finale.

0..1 ***Remarque importante*** : Pour chacune des classes des différentes questions, les méthodes get et set (getters et setters) à définir ne sont pas obligatoirement mentionnées dans les indications données. Vous devrez les ajouter si elles sont nécessaires et non de manière systématique.

1. Classe Obstacle

Définissez la programmation Java d'une classe Obstacle comportant un seul attribut *pénalité* de type int représentant la penalité qui caractérise l'obstacle c'est-à-dire le nombre de points à retrancher au joueur qui le rencontre.
Le constructeur de cette classe possède un paramètre de type int représentant la valeur de l'attribut pénalité. La classe doit aussi fournir une méthode get permettant d'accéder à l'attribut pénalite.

2. Classe Joueur

Définissez la programmation Java de la classe Joueur conformément aux indications données ci-dessous :

- La classe possède 5 attributs :
  - un attribut **nom** de type String
  - un attribut **code** de type String de la forme J suivi d'un numéro correspondant au numéro d'ordre de création du joueur (par exemple : J1 pour le premier joueur, J2 , ..ect…). Il est généré automatiquement lors de la création d'un joueur.
  - un attribut **nbJoueurs** de type int représentant le nombre de joueurs créés.
  - un attribut **nbPoints** est un entier **positif** ou nul qui représente le nombre de points gagnés par le joueur au cours du jeu. Lors de la création du joueur, cet attribut est initialisé à 0.
  - un attribut **listePersos** de type ArrayList<Personnage> représentant la liste des personnages appartenant au joueur.
- Le constructeur possède un paramètre de type String représentant le nom du joueur.
- La méthode **ajouterPersonnage**(p : Personnage) ajoute le personnage p à la liste des personnages du joueur après vérification que le personnage n'est pas déjà attribué à un joueur.
- La méthode **modifierPoints**(nb : int) ajoute la valeur du paramètre nb au nombre de points du joueur (*remarque : nb peut être un nombre négatif*).
- La méthode **peutJouer()** renvoie la valeur vrai si le joueur possède au moins un personnage.
- La méthode **toString**() doit avoir pour résultat une chaîne de la forme suivante :

```
J1 Paul(15 points) avec 2 personnages
```

Ou, dans le cas où le joueur n'a encore aucun personnage :

```
J1 Paul (0 point) aucun personnage
```

3. Classe Personnage

Définissez la programmation Java de la classe **abstraite** Personnage conformément aux indications données suivantes :

- La classe comporte quatre attributs :
  - Un attribut **nom** de type String
  - Un attribut **age** de type int
  - Un attribut **position** est un entier qui représente la position (numéro de case) du personnage dans le tableau des cases du jeu.
  - Un attribut **proprietaire** de type Joueur représentant le propriétaire du personnage.
- Le constructeur possède deux paramètres nom de type String et age de type int.
- La méthode **deplacer**(destination : int , gain : int) modifie la position du personnage en lui attribuant la valeur du paramètre destination et augmente le nombre de points de son joueur propriétaire de la valeur du paramètre gain.
- La méthode **penaliser**(penalite : int) diminue le nombre de points de son joueur propriétaire de la valeur du paramètre penalite.
- La méthode **toString**() renvoie simplement le nom du personnage.
- La méthode **positionSouhaitee()** est abstraite et renvoie un int.

4. Classe Tauren

Définissez la programmation Java de la classe Tauren qui **hérite de la classe personnage** conformément aux indications suivantes :

- L'attribut **taille** représente la taille du Tauren exprimée en mètres.
- Le constructeur possède trois paramètres nom de type String, age de type int et taille de type int.
- La méthode **positionSouhaitee**() renvoie un entier représentant la position que souhaite atteindre le personnage. Cette position est calculée en ajoutant à la position actuelle du personnage un nombre aléatoire compris entre 1 et la valeur de l'attribut taille.

  Pour effectuer ce calcul, vous pourrez utiliser la méthode statique *random* de la classe Math :

  public static double random() qui renvoie un nombre réel x tel que $0 \leq x < 1$.
- La méthode **toString**() doit avoir pour résultat une chaîne de la forme suivante :

  Tauren Hector

5. Classe Humain

Définissez la programmation Java de la classe Humain qui **hérite de la classe personnage** conformément aux indications suivantes :

- L'attribut **nbDeplacements** représente le nombre de déplacements effectués par l'humain. Il est initialisé à 0 lors de la création de l'humain et incrémenté dans la méthode deplacer.
- L'attribut **niveau** représente le niveau de l'humain qui est compris entre 1 et 3. Il est initialisé à 1 lors de la création de l'humain, il prend la valeur 2 lorsque le nombre de déplacements arrive 4 et il prend la valeur 3 lorsque le nombre de déplacements arrive à 6 (modification dans la méthode deplacer).
- Le constructeur possède deux paramètres: nom de type String et age de type int.
- La méthode **deplacer**(destination : int , gain : int) redéfinit la méthode deplacer de Personnage en la complétant pour mettre à jour les attributs niveau et nbDeplacements.

- La méthode **positionSouhaitee**() renvoie un entier représentant la position que souhaite atteindre le personnage. Cette position est calculée en ajoutant à la position actuelle du personnage, le niveau multiplié par le nombre de déplacements.
- La méthode **toString**() doit avoir pour résultat une chaîne de la forme suivante :

Humain Jean

6. Classe Case

Définissez la programmation Java de la classe Case conformément aux indications données ci-dessous.

- La classe comporte trois attributs :
  - Un attribut **gain** correspondant au nombre de points à ajouter au joueur propriétaire du personnage lorsque celui-ci se place sur la case. Ce nombre représente également le nombre de points à retrancher au joueur si son personnage tente de se placer sur la case alors qu'elle est déjà occupée par un autre personnage.
  - Un attribut **perso** de type Personnage représentant le personnage présent sur la case. Cet attribut a la valeur null si aucun personnage n'est présent sur la case.
  - Un attribut **obs** de type Obstacle représentant l'obstacle présent sur la case. Cet attribut a la valeur null si aucun obstacle n'est présent sur la case.
- La classe possède deux constructeurs:
  - un constructeur à deux paramètres: obs de type Obstacle et gain de type int,
  - un constructeur à un paramètre: gain de type int qui permet de créer une case ne comportant pas d'obstacle.
- La méthode **getPenalite()** renvoie 0 si la case ne comporte pas d'obstacle et si la case comporte un obstacle, elle renvoie la penalité caractérisant l'obstacle considéré.
- Les méthodes **placerPersonnage**(perso : Personnage) et **placerObstacle**(obs : Obstacle) positionnent respectivement le personnage perso et l'obstacle obs sur la case.
- La méthode **enleverPersonnage**() affecte la valeur null au personnage perso associé à la case.
- La méthode **estLibre**() renvoie un booléen égal à vrai si la case ne comporte ni obstacle ni personnage
- Les méthodes **sansObstacle()** et **sansPerso**() renvoient un booléen indiquant respectivement l'absence d'obstacle et l'absence de personnage sur la case.
- La méthode **toString**() doit avoir pour résultat une chaîne de la forme suivante :

`Libre (gain = 28)` dans le cas où la case est libre

`Obstacle (penalité = -30)` dans le cas où la case est occupée par un obstacle

`Humain Jean(penalité = -34) ou Tauren Hercule(penalité = -34)` dans le cas où la case est occupée par un personnage (Tauren ou Humain) (la pénalité correspond dans ce cas au montant du gain de la case transformé en nombre négatif)

7. Classe Jeu

Définissez la programmation Java de la classe Jeu conformément aux indications données ci-dessous**:**

- La classe comporte 7 attributs :
  - un attribut **titre** de type String.

- une constante **NB_JOUEUR_MAX** égale à 6 qui correspond au nombre maximum de joueurs pouvant être inscrits à un jeu.
- une constante **NB_CASES** égale à 50 qui correspond au nombre de cases du tableau sur lequel se déplacent les personnages.
- Un attribut **listeJoueurs** de type ArrayList<Joueur> représentant la liste des joueurs inscrits au jeu.
- Un attribut **cases** de type tableau de Case représentant la liste des cases du jeu.
- Un attribut **nbEtapes** de type int correspondant au nombre de déplacements à réaliser par chacun des personnages au cours du déroulement du jeu.
- Un attribut **nbObstacles** de type int correspondant au nombre maximum d'obstacles présents dans le tableau des cases (il s'agit d'un nombre maximum car comme les obstacles sont générés de manière aléatoire, il n'est pas certain que ce nombre soit atteint lors de l'initialisation des cases).
- Un attribut **scoreMax** de type int représentant le score maximum obtenu sur l'ensemble des lancements de jeux.

- Le constructeur possède trois paramètres: String titre de type String, nbEtapes de type int, nbObstacles de type int.
- La méthode **ajouterJoueur**(j : Joueur) ajoute le joueur j à la liste des joueurs inscrits au jeu.
- La méthode **tousLesPersos**() renvoie un ArrayList de Personnage contenant la liste de tous les personnages de tous les joueurs. Cet arrayList est construit en parcourant les listes de personnages de chacun des joueurs du jeu.
- La méthode **initialiserCases**() initialise le tableau des cases en attribuant à chaque case un montant de gains défini par un nombre aléatoire compris entre 1 et le nombre de cases du tableau. Un obstacle est également créé chaque fois que le nombre aléatoire généré est un multiple de 5 (vous utiliserez l'opérateur modulo (reste de la division entière) qui est exprimé en java par l'opérateur % : 10%3=1 par exemple). Chaque obstacle est créé avec une pénalité égale au double du nombre aléatoire généré et il est placé dans la case considérée. Le nombre effectif d'obstacles ne devra toutefois pas dépasser le nombre maximum d'obstacles attribué au jeu (attribut nbObstacles).
- La méthode **lancerJeu()** commence par positionner l'ensemble des personnages (renvoyé par la méthode **tousLesPersos**) dans le tableau des cases. Ainsi, le premier personnage est placé sur la case 0 si celle-ci ne comporte pas d'obstacle, si elle comporte un obstacle, on le place dans la case 1, ect… On passe ensuite au personnage suivant et ainsi de suite.

Le jeu démarre alors.

Lors de chaque étape, chacun des personnages tente de se déplacer. Si la position souhaitée par le personnage correspond à une case libre, le personnage est effectivement déplacé et le joueur associé gagne des points. Si la position souhaitée correspond à une case avec obstacle, le personnage n'est pas déplacé et le joueur associé se voit attribuer une pénalité. De même si la case est déjà occupée par un autre personnage, le joueur associé se voit attribuer une pénalité égale au montant des gains associé à la case. Si la position souhaitée dépasse le numéro de la dernière case du tableau, on considère que le joueur souhaite atteindre la dernière case du tableau.

La méthode se termine lorsque toutes les étapes se sont déroulées. Elle affiche alors les résultats.

- La méthode **afficherCases()** affiche le tableau des cases sous la forme suivante

```
Case 0 : Humain Jean (penalité = -7)
```

```
Case 1 : Obstacle (penalité = -70)
Case 2 : Obstacle (penalité = -20)
Case 3 : Libre (gain = 37)
Case 4 : Libre (gain = 31)
Case 5 : Libre (gain = 37)
Case 6 : Libre (gain = 33)
Case 7 : Libre (gain = 49)
Case 8 : Tauren Hercule (penalité = -43)
...
```

- La méthode **afficherParticipants()** affiche la liste des joueurs inscrits au jeu sous la forme suivante

```
LISTE DES JOUEURS
------------------------
J1 Paul(253 points) avec 2 personnages
------------------------
J2 Lucien(240 points) avec 2 personnages
```

- La méthode **afficherResultats()** affiche la liste des joueurs inscrits au jeu en invoquant la méthode **afficherParticipants**() puis affiche les résultats du jeu sous la forme suivante :

```
JEU AtelierPOO
*******************************************
RESULTATS
Le gagnant est Paul avec 253 points
Record battu : Ancien score maximum 134
```

8. Classe TestJeu

Définissez la programmation Java de la classe **TestJeu** comportant une méthode main réalisant la répétition (5 fois) les actions suivantes :

- Créer un jeu ayant pour titre *AtelierPOO* comportant 4 étapes et 10 obstacles maximum.
- Créer un joueur nommé Paul et lui attribuer deux personnages :
  - un tauren de nom Hector agé de 15 ans et de taille 10
  - un humain nommé Jean agé de 10 ans.
- Créer un joueur nommé Lucien et lui attribuer deux personnages :
  - un humain de nom Marie agé de 10 ans
  - un tauren nommé Hercule agé de 20 ans et de taille 5.
- Inscrire les deux joueurs au jeu AtelierPOO
- Lancer le jeu
- Afficher la liste des participants et le résultat comportant l'affichage du score max.

8. Correction de l'exercice sur le zoo

1. **Classe abstraite** `Animal` avec un attribut encapsulé pour le nom et l'âge.
2. **Attribut de classe** `nombre_animaux` qui compte le nombre total d'animaux.
3. Les classes dérivées **Chien**, **Chat**, et **Lion** implémentent chacune la méthode `parler()` de manière différente (polymorphisme).
4. La classe **Zoo** gère une collection d'animaux et affiche des informations sur chacun.
5. Utilisation de la méthode `@property` pour gérer l'encapsulation et les accès à l'attribut `age`.

Exemple de Code Attendu :

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    # Attribut de classe pour compter le nombre d'animaux
    nombre_animaux = 0

    def __init__(self, nom, age):
        self.__nom = nom  # Encapsulation (attribut privé)
        self.__age = age  # Encapsulation (attribut privé)
        Animal.nombre_animaux += 1

    # Getter pour le nom
    def get_nom(self):
        return self.__nom

    # Getter et Setter pour l'âge avec @property
    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        if age >= 0:
            self.__age = age
        else:
            print("L'âge doit être positif.")

    # Méthode abstraite
    @abstractmethod
    def parler(self):
        pass

class Chien(Animal):
    def parler(self):
        return f"{self.get_nom()} dit Woof!"

class Chat(Animal):
```

```
    def parler(self):
        return f"{self.get_nom()} dit Meow!"

class Lion(Animal):
    def parler(self):
        return f"{self.get_nom()} rugit!"

class Zoo:
    def __init__(self):
        self.animaux = []

    def ajouter_animal(self, animal):
        self.animaux.append(animal)

    def afficher_informations(self):
        for animal in self.animaux:
            print(f"Nom : {animal.get_nom()}, Âge : {animal.age}, Son : {animal.parler()}")
        print(f"Nombre total d'animaux : {Animal.nombre_animaux}")

# Exemple d'utilisation
zoo = Zoo()

chien = Chien("Rex", 5)
chat = Chat("Felix", 3)
lion = Lion("Simba", 7)

zoo.ajouter_animal(chien)
zoo.ajouter_animal(chat)
zoo.ajouter_animal(lion)

# Afficher les informations des animaux dans le zoo
zoo.afficher_informations()
```