7.2 KLASSEN UND OBJEKTE

EINFÜHRUNG

.

Du hast bereits Bekanntschaft mit wichtigen Konzepten der Objektorientierten Programmierung gemacl und gemerkt, dass du ohne OOP in Python kaum Computergames schreiben kannst. Es ist daher wichtig dass du etwas systematischer die Begriffe der OOP und ihre Implementierung in Python kennen lern [mehr...]

PROGRAMMIERKONZEPTE:*Vererbung, Klassenhierarchie, Überschreiben, Is-a-Relation, Mehrfachvererbung*

INSTANZVARIABLEN

Tiere eignen sich hervorragend, also Objekte modelliert zu werden. Du definierst zuerst eine Klasse *Anima* die das entsprechende Tierbild im Hintergrund des Gameboard darstellt. Bei der Erzeugung eines Objek dieser Klasse übergibst du daher dem Konstruktor den Dateipfad auf das Tierbild, damit die Method *showMe()* soll das Bild anzeigen kann. Sie verwendet dabei Zeichnungsmethoden der Klass *GGBackground*.

Der Konstruktor, der den Dateipfad erhält, muss ihn als Initialisierungswert in einer Variablen speicher damit alle Methoden darauf zugreifen können. Eine solche Variable ist ein Attribut oder eine Instanzvariabl der Klasse. In Python erhalten Instanzvariablen den Prefix *self* und werden bei der ersten Zuweisung eine Werts erzeugt.

Wie du bereits weist, hat der Konstruktor den speziellen Namen __init__ (mit zwei vor- und nachgestellte Underlines). Der Konstruktor sowie alle Methoden müssen self als ersten Parameter aufweisen, was ma gerne vergisst.

Du definierst also den Konstruktor

```
def __init__(self, imgPath):
```

sowie eine Methode

```
def showMe(self, x, y):
```

Hast du einmal ein Tierobjekt myAnimal mit

```
myAnimal = Animal(bildpfad)
```

erzeugt, so rufst du diese Methode mit

```
myAnimal.showMe(x, y)
```

auf, also ohne den Parameter self. Die OOP macht vor allem dann einen Sinn, wenn du mehrere Objek derselben Klassen verwendest. Um dies hautnahe zu erleben, soll in deinem Programm bei jede Mausklick ein neues Tier entstehen.

```
from gamegrid import *

# ---------------- class Animal ----------------
class Animal():
    def __init__(self, imgPath):
        self.imagePath = imgPath  # Instance variable
    def showMe(self, x, y):  # Method definition
         bg.drawImage(self.imagePath, x, y)

def pressCallback(e):
```

```
    myAnimal = Animal("sprites/animal.gif") # Object creation
    myAnimal.showMe(e.getX(), e.getY())  # Method call

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

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Eigenschaften oder Attribute eines Objekts werden als Instanzvariablen definiert. Sie haben für jede Objekt der Klasse individuelle Werte. Der Zugriff auf Instanzvariablen innerhalb der Klasse erfolgt durc Vorstellen von self. Von ausserhalb der Klasse kann man durch Voranstellen des Instansnamens darau zugreifen, wie z. B. myInstance.attribut.

Einer Klasse stehen aber auch die Variablen und Funktionen des Programm-Hauptteils zur Verfügung beispielsweise alle Methoden der Klasse GameGrid und mit *bg* der Background des Spielfensters. D Methoden können sogar eine Variable des Hauptteils verändern, falls sie in der Methode als glob deklariert wird.

Wenn das Objekt keine Initialisierungen benötigt, so kann die Definition des Konstruktors auc weggelassen werden. Statt das Spritebild dem Konstruktor zu übergeben, verwendest du im folgende Programm die Variable *imagePath* und kannst damit auf den Konstruktor verzichten.

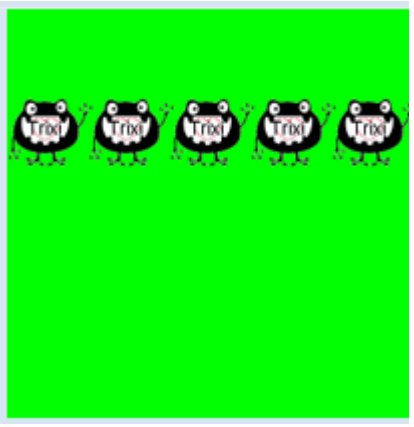
```
from gamegrid import *
import random

# ---------------- class Animal ----------------
class Animal():
    def showMe(self, x, y):
         bg.drawImage(imagePath, x, y)

def pressCallback(e):
    myAnimal = Animal()
    myAnimal.showMe(e.getX(), e.getY())

imagePath = "sprites/animal.gif"
makeGameGrid(600, 600, 1, False, mousePressed = pressCallback)
setBgColor(Color.green)
show()
doRun()
bg = getBg()
```

VERERBUNG, METHODEN HINZUFÜGEN

Durch Klassenableitung oder Vererbung erstellst du eine Klassenhierarchie und kannst damit eine bestehenden Klasse zusätzliche Eigenschaften und Verhalten hinzufügen. Objekte der abgeleiteten Klass sind automatisch auch Objekte der übergeordneten Klasse (auch **Ober-, Basis**- oder **Superklass** genannt) und können daher

alle Eigenschaften und Methoden der übergeordneten Klasse verwenden, als ob sie in der abgeleiteten Klasse selbst definiert wären.

Beispielsweise soll ein Haustier ein Tier sein, dass zusätzlich noch einen Namen hat, den es mit *tell()* ausschreiben soll. Du definierst daher eine Klasse *Pet*, die von *Animal* abgeleitet ist. Da du den Tiernamen für jedes Haustier individuell bei seiner Erzeugung festlegen willst, übergibst du ihn als Initialisierungswert dem Konstruktor von *Pet*, der ihn in einer Instanzvariablen abspeichert.




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

# ---------------- class Animal ----------------
class Animal():
    def __init__(self, imgPath):
        self.imagePath = imgPath
    def showMe(self, x, y):
         bg.drawImage(self.imagePath, x, y)

# ---------------- class Pet ----------------
class Pet(Animal):   # Derived from Animal
    def __init__(self, imgPath, name):
        self.imagePath = 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)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

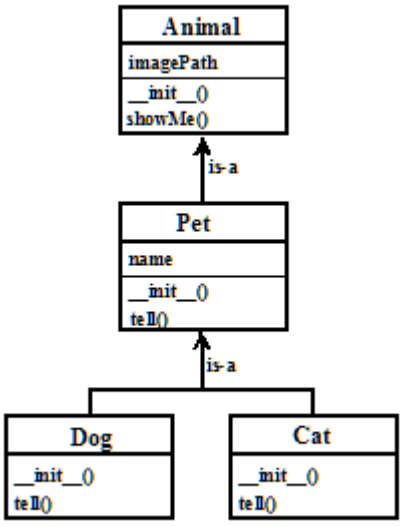
Wie du siehst, kannst du *myPet.showMe()* aufrufen, obschon *showMe()* in der Klasse *Pet* gar nicht definie ist, denn ein Haustier **ist-auch-ein** Tier. Man nennt die Beziehung von *Pet* und *Animal* daher eine **is-a-Relation**.

Da die Instanzvariable *imagePath* im Konstruktor von *Animal* definiert ist, kannst du die Zei *self.imagePath = imgPath* im Konstruktor von *Pet* durch *Animal.__init__(self, imgPath)* ersetzen, wodurc die Basisklasse *Animal* initialisiert wird.

Die Basisklassen werden bei den abgeleiteten Klassen in eine Klammer hinter den Klassennamen gesetz In Python kann man eine Klasse auch aus mehreren Basisklassen ableiteten (**Multiple Inheritance**).

KLASSENHIERARCHIE, METHODEN ÜBERSCHREIBEN

In einer abgeleiteten Klasse können Methoden der Basisklasse auch verändert werden, indem man sie mit dem gleichen Namen und der gleichen Parameterliste neu definiert. Willst du Hunde modellieren, die bei *tell()* auch noch bellen, so leitest du die Klasse *Dog* von *Pet* ab und überschreibst die Methode *tell()*. Analog kannst du eine Katze miauen lassen, indem du eine Klasse *Cat* von *Pet* ableitest und dort ebenfalls *tell()* überschreibst.



Die vier Klassen können anschaulich in einer **Klassendiagramm** aufgezeichnet werden. Darin wird die is-a-Relation besonders deutlich [mehr...].

Im Klassendiagramm werden die Klassen als Rechteckbox dargestellt, in die du zuerst den Klassennamen schreibst. Mit einer horizontalen Trennungslinie getrennt folgen als nächstes die Instanzvariablen und dann, angeführt durch den Konstruktor, die Methoden der Klasse. Die Klassenhierarchie wird durch eine geschickte Anordnung und mit Verbindungspfeilen anschaulich gemacht.

```
from gamegrid import *

# ---------------- class Animal ----------------
class Animal():
    def __init__(self, imgPath):
        self.imagePath = imgPath
    def showMe(self, x, y):
         bg.drawImage(self.imagePath, x, y)

# ---------------- class 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))

# ---------------- class 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))

# ---------------- class 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)  # Overriden method is called

rex = Dog("sprites/dog.gif", "Rex")
rex.showMe(100, 300)
rex.tell(200, 330)  # Overriden method is called

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

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Durch das Überschreiben von Methoden kann man in der abgeleiteten Klassen das Verhalten de Basisklasse verändern.

Beim Aufruf von Methoden derselben Klasse oder der Basisklasse muss *self* vorgestellt werden. In de Parameterliste wird *self* aber nicht übergeben.

Manchmal möchte man in einer überschriebenen Methode die gleichlautende Methode der Basisklass verwenden. Um diese aufzurufen, muss man den Klassennamen der Basisklasse voranstellen und in de Parameterliste *self* ebenfalls übergeben [mehr...].

Diese Regel gilt auch für den Konstruktor: Wird im Konstruktor der abgeleiteten Klasse der Konstruktor de Basisklasse verwendet, so muss dieser durch Vorstellen des Klassennamens der Basisklasse und mit der Parameter *self* aufgerufen werden. Beispielsweise:

```
class BaseClass:

    def __init__(self, a):
        self.a = a

class ChildClass(BaseClass):

    def __init__(self, a, b):
        # die Initialisierung der Instanzvariablen 'a'
        # erfolgt durch den Konstruktor der Basisklasse
        BaseClass.__init__(self, a)
        self.b = b
```

TYPENBEZOGENER METHODENAUFRUF: POLYMORPHISMUS

Eine etwas schwieriger zu verstehende, aber besonders wichtige Eigenschaft von objektorientierte Programmiersprachen ist der Polymorphismus Darunter versteht man den Aufruf von überschriebene

Methoden, wobei der Aufruf automatisch der Klassenzugehörigkeit angepasst wird. An einem einfache Beispiel erlebst du, was damit gemeint ist.

Du verwendest mit den vorher definierten Klassen eine Liste *animals*

```
animals = [Dog(), Dog(), Cat()]
```

in der sich zwei Hunde und eine Katze befinden. Beim Durchlaufen der Liste und Aufruf von *tell()* mit

```
for animal in animals:
    animal.tell()
```

tritt eine Schwierigkeit auf, denn es gibt ja drei verschiedene Methoden von *tell()*, nämlich je eine in de Klassen *Pet*, *Dog* und *Cat*. Der Computer kann diese Vieldeutigkeit auf drei Arten auflösen. Er kann

1. eine Fehlermeldung abgeben,
2. das *tell()* der Basisklasse *Pet* aufrufen,
3. herausfinden, um welche Sorte von Pets es sich handelt und das entsprechende *tell()* aufrufen.

In einer polymorphen Programmiersprache wie Python gilt das letzte und beste Verhalten.

```
from gamegrid import *
from soundsystem import *

# ---------------- class Animal ----------------
class Animal():
    def __init__(self, imgPath):
        self.imagePath = imgPath
    def showMe(self, x, y):
         bg.drawImage(self.imagePath, x, y)

# ---------------- class 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))

# ---------------- class Dog ----------------
class Dog(Pet):
    def __init__(self, imgPath, name):
         Pet.__init__(self, imgPath, name)
    def tell(self, x, y): # Overridden
         Pet.tell(self, x, y)
         openSoundPlayer("wav/dog.wav")
         play()

# ---------------- class Cat ----------------
class Cat(Pet):
    def __init__(self, imgPath, name):
        Pet.__init__(self, imgPath, 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()
```

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

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

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Der Polymorphismus sorgt dafür, dass bei überschriebenen Methoden die Klassenzughörigkeit entscheide welche Methode aufgerufen wird. Da in Python die Zugehörigkeit zu Klassen sowieso erst zu Laufze festgelegt wird, ist der Polymorphismus eine Selbstverständlichkeit.

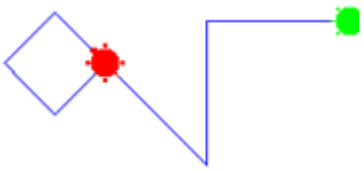
Diese dynamische Datenbindung von Python nennt sich auch ***Ententest*** oder ***Duck-Typing***, gemäss der Zitat, das James Whitcomb Riley (1849 - 1916) zugeschrieben wird:

"Wenn ich einen Vogel sehe, der wie eine Ente läuft, wie eine Ente schwimmt und wie eine Ente schnatter dann nenne ich diesen Vogel eine Ente."

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

AUFGABEN

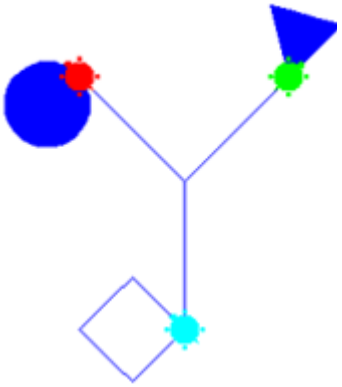
1. Definiere eine aus der Klasse *Turtle* abgeleitete Klasse *TurtleKid*, welche mit *shape()* ein Quadra zeichnet. Der folgende Hauptteil soll funktionieren:



```
tf = TurtleFrame()
# john ist eine Turtle
john = Turtle(tf)
# john kennt alle Befehle der Turtle
john.setColor("green")
john.forward(100)
john.right(90)
john.forward(100)

# laura ist ein TurtleKid, aber auch eine Turtle
# laura kennt alle Befehle der Turtle
laura = TurtleKid(tf)
laura.setColor("red")
laura.left(45)
laura.forward(100)
# laura kennt aber auch den neuen Befehl
laura.shape()
```

2. Definiere zwei aus *TurtleKid* abgeleitete Klassen *TurtleBoy* und *TurtleGirl*, welche *shape()* s überschreiben, dass ein *TurtleBoy* ein gefülltes Dreieck und *ein TurtleGirl* einen gefüllten Kre zeichnet Der folgende Hauptteil soll funktionieren:

```
tf = TurtleFrame()

aGirl = TurtleGirl(tf)
aGirl.setColor("red")
aGirl.left(45)
aGirl.forward(100)
aGirl.shape()

aBoy = TurtleBoy(tf)
aBoy.setColor("green")
aBoy.right(45)
aBoy.forward(100)
aBoy.shape()

aKid = TurtleKid(tf)
aKid.back(100)
aKid.left(45)
aKid.shape()
```



3. Zeichne das Klassendiagramm zu Aufgabe 2

ZUSATZSTOFF

STATISCHE VARIABLEN UND STATISCHE METHODEN

Klassen können auch dazu verwendet werden, zusammengehörende Variablen oder Funktionen z gruppieren und damit den Code übersichtlicher zu machen. Beispielsweise kannst du die wichtigste physikalischen Konstanten in der Klasse *Physics* zusammenfassen. Man nennt Variablen, die i Klassenkopf definiert werden, **statische Variablen** und ruft sie durch Vorstellen des Klassennamens au Es ist also im Gegensatz zu Instanzvariablen nicht nötig, eine Instanz der Klasse zu erzeugen.

```
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)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Eine Sammlung von zusammengehörenden Funktionen kannst du ebenfalls gruppieren, indem du sie a statische Methoden in einer aussagekräftig bezeichneten Klasse definierst. Diese Methoden kannst d dann durch Vorstellen des Klassennamens direkt verwenden, ohne dass du eine Instanz der Klass erstellen musst.

Um eine Methode statisch zu machen, muss man vor die Definition @staticmethod schreiben.

```
# ---------------- 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)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Statische Variable (im Unterschied zu Instanzvariablen auch **Klassenvariablen** genannt) gehören zu de Klasse als Ganzes und haben im Gegensatz zu Instanzvariablen für alle Objekte der Klasse den gleiche Wert. Sie können mit vorgestelltem Klassennamen gelesen und verändert werden.

Eine typische Anwendung von statischen Variablen ist ein *Instanzenzähler,* also eine Variable, welche di Anzahl erzeugter Objekte der betreffenden Klasse zählt.

Zusammengehörende Funktionen können als statische Methoden einer sinnvoll bezeichneten Klass gruppiert werden. Bei der Definition muss die Zeile @staticmethod (**function decorator** genann vorgestellt werden