7.1 OBJEKTE ÜBERALL

EINFÜHRUNG

Im täglichen Leben bist du von einer Vielzahl von verschiedenen Objekten umgeben. Da Software oft di Wirklichkeit modellmässig abbildet, ist es naheliegend, auch in der Informatik Objekte einzuführen. Ma spricht dann von **Objektorientierter Programmierung (OOP)**. Das Konzept der OOP hat sich seit zw Jahrzehnten in der Informatik als derart wegweisend erwiesen, dass es praktisch in allen heut entwickelten Softwaresystemen angewendet wird [mehr...]. Im Folgenden lernst du die wichtigste Konzepte der OOP kennen, damit du an diesem Hype teilhaben kannst.

Du hast bereits die Turtle als Objekt kennen gelernt. Eine Turtle besitzt **Eigenschaften** (sie hat ein bestimmte Farbe, befindet sich an einer gewissen Position und hat eine bestimmte Blickrichtung) un **Fähigkeiten** (sie kann sich vorwärts bewegen, sich drehen, usw.). In der OOP werden Objekte m ähnlichen Eigenschaften und Fähigkeiten in **Klassen** eingeteilt. Die Turtleobjekte gehören zur *Klass Turtle*, man sagt auch, sind **Instanzen** der *Klasse Turtle*. Um ein Objekt zu erzeugen, muss man zuerst ein **Klasse definieren** oder wie bei Turtles eine bereits vordefinierte Klasse verwenden.

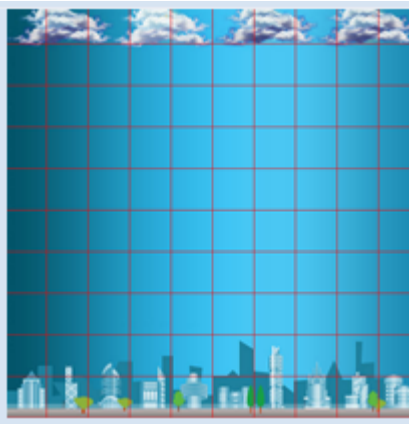
Beim Programmieren nennt man die Eigenschaften auch **Attribute** oder **Instanzvariablen**, die Fähigkeite auch **Operationen oder Methoden**. Es handelt sich um Variablen und Funktionen, ausser dass sie eine bestimmten Klasse angehören und damit in der Klasse "gekapselt" sind. Um sie ausserhalb der Klasse z verwenden, muss man eine Klasseninstanz und den Punktoperator voranstellen.

PROGRAMMIERKONZEPTE:*Grundbegriffe: Klasse, Objekt (Instanz), Eigenschaft, Fähigkeit, Attribut/Instanzvariable, Methode, Ableitung, Basis-/Superklasse, Konstrukto*

EIN ANPASSUNGSFÄHIGES SPIELFENSTER

Ohne OOP ist es nicht möglich, mit vernünftigem Aufwand ein Computergame zu erstellen, denn di Spielfiguren und andere Gegenstände des Gameboards sind offensichtlich Objekte. Das Gameboard ist ei rechteckiges Bildschirmfenster und wird durch die **Klasse GameGrid** aus der Gamelibrary **JGameGri** modelliert. TigerJython stellt dir eine Instanz beim Aufruf von makeGameGrid() zur Verfügung und m show() wird das Fenster angezeigt. Dabei kannst du das Aussehen des Spielfensters über Parameterwert deinen Wünschen anpassen. Mit `makeGameGrid(10, 10, 60, Color.red "sprites/town.jpg", False)`

wird ein quadratisches Spielfenster angezeigt, dass 10 horizontale und 10 vertikale Zellen der Gr⊘sse 60 Pixel aufweist. Du erkennst rote Gitterlinien und ein Hintergrundbild town.jpg. (Der letzte Parameter bewirkt, dass der Navigationsbalken unterdr⊘ckt wird, der hier nicht benötigt wird.)



```
from gamegrid import *

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

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

MEMO

Die Methoden der Klasse *GameGrid* stehen dir mit *makeGameGrid()* als Funktionen zur Verfügung. D kannst aber auch selbst eine Instanz erzeugen und die Methoden mit dem Punktoperator aufrufen.

```
from gamegrid import *

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

Das Spielfenster ist aus quadratischen Zellen aufgebaut, wobei die Zellengrösse (60 pixel) und die Anzal 10 horizontaler und vertikaler Zellen angegebenen werden. Damit auch die rechte und untere Gitterlini angezeigt werden, besitzt das Fenster eine Grösse von 601 x 601 pixel. Dies entspricht der (minimaler Grösse des Hintergrundbildes

Der letzte boolesche Parameter bestimmt, ob eine Navigationsleiste erscheint.

KLASSENDEFINITION MIT ABLEITUNG

Bei der Definition einer Klasse kannst du entscheiden, ob deine neue Klasse ganz eigenständig ist ode aus einer bereits vorhandenen Klasse **abgleitet** wird. In der abgeleiteten Klasse stehen dir all Eigenschaften und Fähigkeiten der **Oberklasse** (auch **Basisklasse** oder **Superklasse** genannt) zu Verfügung. Anschaulich sagt man, dass die abgeleitete Klasse Eigenschaften und Fähigkeiten **erbt**.

In der Gamelibrary *JGameGrid* werden Spielfiguren **Actors** genannt und sind Instanzen der vordefinierte *Klasse Actor*. Willst du also eine eigene Spielfigur verwenden, so **definierst** du eine Klasse, die **aus Acto abgeleitet** ist.

Deine Klassendefinition beginnt mit dem Schlüsselwort **class**. Es folgt der beliebig wählbare Klassenname und ein rundes Klammerpaar. Dort schreibst du den Namen einer oder mehrerer Klassen hin, von denen d deine Klasse ableitest. Da du die Spielfigur von *Actor* ableiten willst, gibst du diesen Klassennamen an.

Die Klassendefinition enthält die Definition der **Methoden**, die wie normale Funktionen definiert werden, m dem Unterschied, dass sie **obligatorisch** den Parameter **self** als ersten Parameter aufweisen müssen. M diesem Parameter kannst du auf andere Methoden und Instanzvariablen der eigenen Klasse und ihre Basisklasse zugreifen.

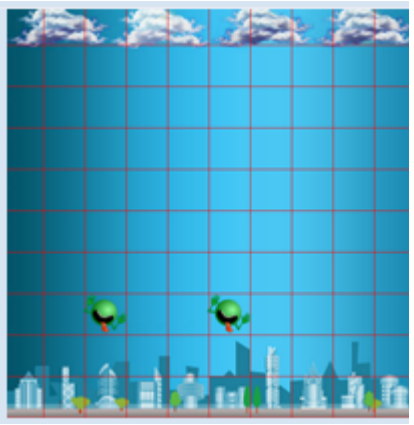
Die Liste der Methodendefinitionen beginnt üblicherweise mit der Definition einer **speziellen Methode** m dem Namen __init__ (zwei vor- und nachgestellte Underlines). Diese nennt man **Konstruktor** (in Pytho auch **Initializer** genannt) und sie wird automatisch dann aufgerufen, wenn ein Objekt der Klasse erzeug wird. In unserem Fall rufst du im Konstruktor von *Alien* den Konstruktor der Basisklasse *Actor* auf, welche du den Pfad zum Spritebild übergibst.

Als nächstes definierst du die Methode **act()**. Diese spielt für die Gameanimation eine zentrale Rolle, den sie wird vom Gamemanager in jedem Simulationszyklus automatisch aufgerufen. Dies ist ein besonde intelligenter Trick, damit du dich nicht mit einer Wiederholstruktur selbst um die Animation kümmern musst

In act() legst du fest, was die Spielfigur in jedem Simulationszyklus machen soll. Als Demonstratio bewegst du sie hier lediglich mit move() in die nächste Zelle. Da *move()* eine Methode der Basisklass *Actor* ist, musst du sie mit vorgestelltem *self* aufrufen.

Hast du einmal deine Klasse *Alien* definiert, so **erzeugst** du ein Alienobjekt unter Aufruf des Klassennamens und weist es einer Variablen zu. Typisch für die OOP ist es, dass du selbstverständlich beliebig viele Aliens erzeugen kannst. Wie im täglichen Leben haben diese eine eigene **Individualität**, "wissen" also durch ihre *act()*-Methode, wie sie sich bewegen müssen.

Um die erzeugten Aliens ins Gameboard einzufügen, verwendest du addActor(), wobei du mit *Location()* die Zellenkoordinaten angeben musst (die Zelle mit den Koordinaten (0,0) ist oben links, x nimmt nach rechts, y nach unten zu). Um den Simulationszyklus zu starten, rufst du schliesslich doRun() auf.



```
from gamegrid import *

# ---------------- class Alien ----------------
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()
```

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

MEMO

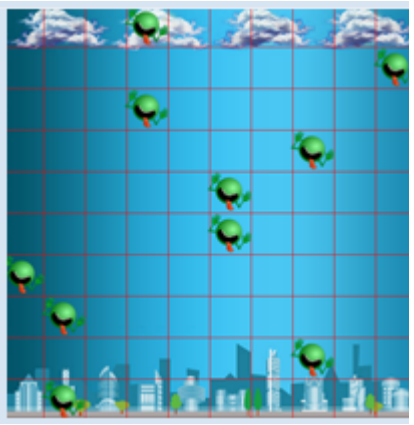
Eine Klassendefinition beginnt mit dem Schlüsselwort **class** und **kapselt** die Methoden un Instanzvariablen der Klasse. Der Konstruktor mit dem Namen **__init__** wird bei der Erzeugung eine Objekts automatisch aufgerufen. Um ein Objekt (eine Instanz) zu erzeugen, verwendest du de Klassennamen und übergibst ihm die Parameterwerte, welche __init__ erhalten soll.

Alle Spielfiguren werden aus der Klasse *Actor* abgeleitet. In der Methode act() definierst du, was d Spielfigur in jedem Simulationszyklus machen soll.

Mit addActor() fügst du eine Spielfigur in das Gameboard, wobei du seine Startposition (Location) und sein Startrichtung (in Grad) angibst (0 gegen Osten, positiv in Uhrzeigersinn).

ANGRIFF DER ALIENS

Du erkennst die Stärke und Eleganz des objektorientieren Programmierparadigmas daran, dass du das Gameboard mit wenigen Programmzeilen mit vielen vom Himmel fallenden Aliens bevölkern kannst. Dazu modellierst du im Hauptteil mit einer Wiederholschleife ein Alienraumschiff, das alle 0.2 s einen neuen Alien an zufälliger Stelle der obersten Gitterzeile aussetzt.



```
from gamegrid import *
from random import randint

# ---------------- class Alien ----------------
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)
show()
doRun()

while not isDisposed():
    alien = Alien()
    addActor(alien, Location(randint(0, 9), 0), 90)
    delay(200)
```

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

MEMO

Eine Endlosschleife im Hauptteil des Programms sollte mit isDisposed() darauf testen, ob das Gamefenste geschlossen wurde, damit das Programm korrekt beendet wird.

Achtung: Es ist manchmal nötig, TigerJython zu schliessen und wieder zu öffnen, damit gleichnamige aber veränderte Sprite- oder Hintergrundbilder geladen werden.

SPACEINVADER LIGHT

In deinem ersten selbst geschriebenen Computergame soll der Spieler versuchen, eine Alien-Invasion z bekämpfen, indem er die angreifenden Aliens mit Mausklicks entfernt. Jeder in der Stadt gelandete Alie führt zu einem Minuspunkt.

Für die Mausunterstützung fügst du einen Maus-Callback mit dem Namen *pressCallback* ein ur registrierst ihn als benannten Parameter *mousePressed*. Im Callback holst du dir zuerst aus dem Even Parameter e die Zellenposition des Mausklicks. Befindet sich in dieser Zelle ein Actor, so kriegst du ihn m getOneActorAt(), ist die Zelle leer, so liefert der Aufruf None. removeActor() entfernt den Actor aus der Gameboard.

```
from gamegrid import *
from random import randint
```

```
# ---------------- class Alien ----------------
class Alien(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/alien.png")

    def act(self):
        self.move()

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(800)
show()
doRun()

while not isDisposed():
    alien = Alien()
    addActor(alien, Location(randint(0, 9), 0), 90)
    delay(1000)
```

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

MEMO

Da *act()* in jeder Simulationsperiode einmal aufgerufen wird, ist die Periodendauer für di Ablaufgeschwindigkeit des Games verantwortlich. Der Standardwert der Simulationsperiode ist 200 ms. Si kann mit setSimulationPeriod() auf einen anderen Wert eingestellt werden.

Das Gameboard wird in jedem Simulationszyklus einmal neu aufgebaut (gerendert), eine Änderung de Spielsituation wird daher erst zu diesem Zeitpunkt sichtbar. Willst du bei einem Mausklick die neu Situation sofort anzeigen, so kannst du das Rendern mit refresh() manuell ausführen.

AUFGABEN

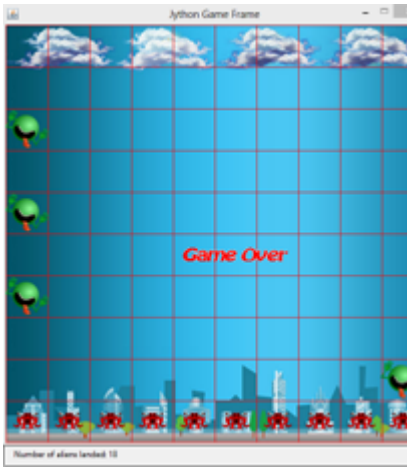
1. Erstelle mit einem Bildeditor ein eigenes Hintergrundbild. Füge es in das Verzeichnis sprites i Verzeichnis, in dem sich dein Programm befindet (oder in *<userhome>/gamegrid/sprites*) oder geb den voll qualifizierten Dateipfad an.

2. Füge mit *addStatusBar(30)* eine 30 pixel hohe Statusbar an und schreibe dort mit *setStatusText()* d Anzahl Aliens aus, die trotz der Abwehr in der Stadt landen konnten.

3. Die gelandeten Aliens sollen nicht einfach verschwinden, sondern sich an der Landestelle in ein andere Form ("sprites/alien_1.gif" oder eigenes Bild) umwandeln und in der Landestelle verharre (Anleitung: Mit *removeSelf()* kannst du einen alten Alien entfernen und mit *addActor()* einen neue Actor an derselben Stelle erzeugen.)

4*.

Die gelandeten Aliens melden melden an das Alienraumschiff, wo sie gelandet sind, so dass neue Aliens nur noch in "freien" Spalten abspringen. Sobald alle Spalten besetzt sind, ist das Spiel mit einer Anzeige von "GameOver" beendet ("sprites/gameover.gif"). (Anleitung: Der Gamemanager kann mit *doPause()* angehalten werden.)



5*. Erweitere das Game nach deinen eigenen Ideen.