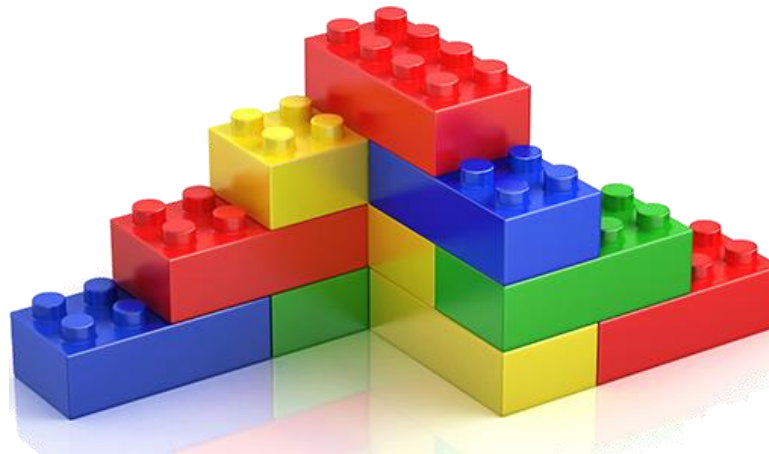




**UNIVERSITÀ  
DI PARMA**

# **composizione** **informatica e laboratorio di programmazione**

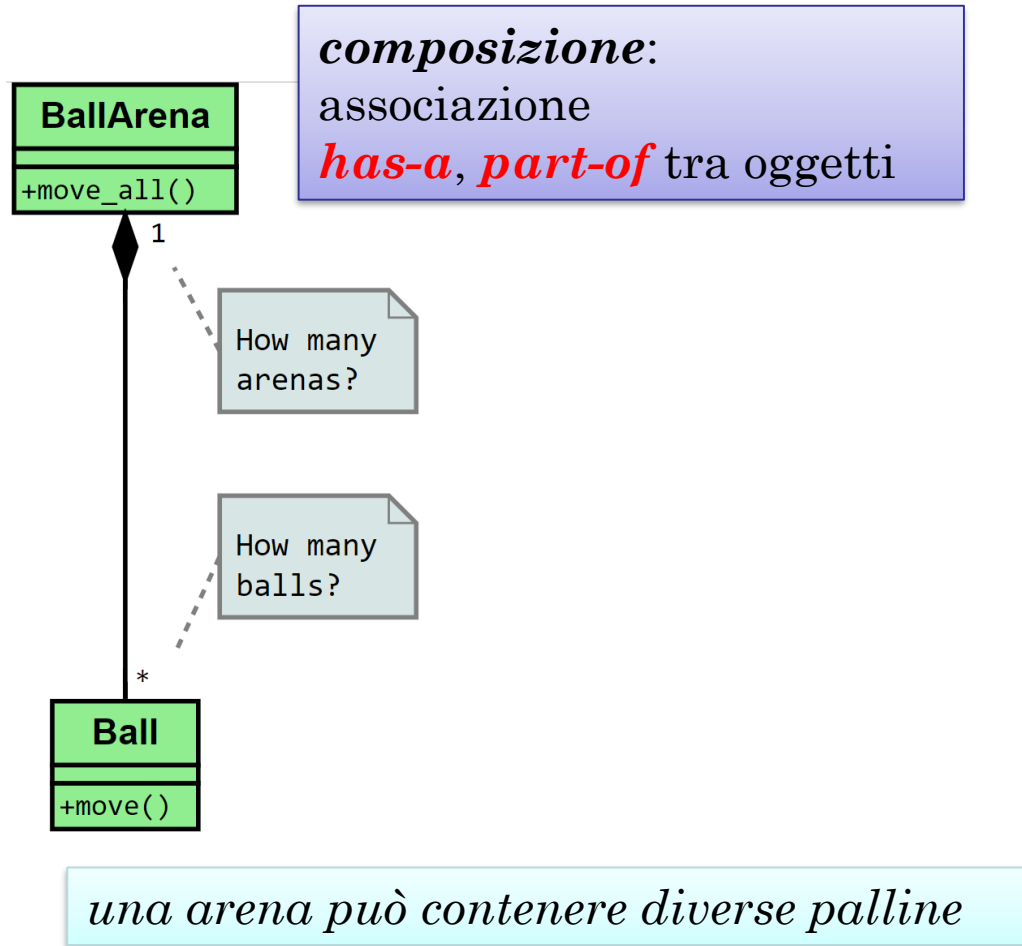


```
import g2d
from sl04_01_Ball import Ball, ARENA_W, ARENA_H

def update():
    g2d.fill_canvas((255, 255, 255)) # BG
    for b in balls:
        b.move()
        g2d.draw_rect((127, 127, 127), b.position()) # FG

balls = [Ball(40, 80), Ball(80, 40), Ball(120, 120)]
g2d.init_canvas((ARENA_W, ARENA_H))
g2d.main_loop(update, 1000 // 30) # Millis
```

[http://www.ce.unipr.it/brython/?p3\\_oop\\_balls.py](http://www.ce.unipr.it/brython/?p3_oop_balls.py)

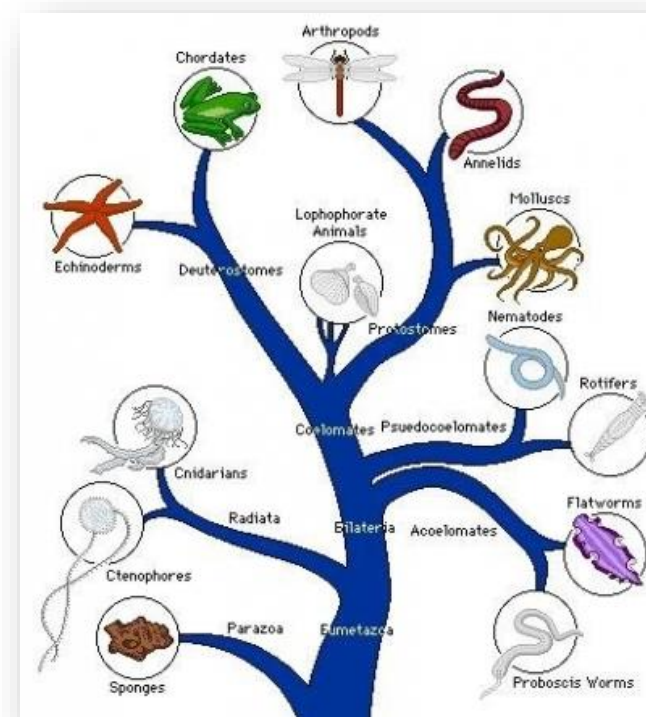


```
class BallArena: # ...
    def __init__(self):
        self._balls = []
    def add(self, b: Ball):
        self._balls.append(b)
    def move_all(self):
        for b in self._balls:
            b.move()
            g2d.draw_rect((127, 127, 127), b.position())

def update():
    g2d.fill_canvas((255, 255, 255)) # BG
    arena.move_all()

arena = BallArena()
arena.add(Ball(40, 80))
arena.add(Ball(80, 40)) # ...
g2d.init_canvas((ARENA_W, ARENA_H))
g2d.main_loop(update, 1000 // 30) # Millis
```

- relazione **is-a**
  - **classificazione**, es. in biologia
    - vertebrati **sottoclasse** di animali
    - mammiferi **sottoclasse** di vertebrati
    - felini **sottoclasse** di mammiferi
    - gatti **sottoclasse** di felini
- relazione **is-a tra classi**: ogni sottoclasse...
  - **eredita** le caratteristiche della classe base
  - ma introduce delle **specializzazioni**



- definiremo una *classe base* come *interfaccia astratta*
  - es. Animale:
- tutti gli animali fanno un verso (*interfaccia*)
- ogni animale fa un verso diverso (*polimorfismo*)

```
class Animale:  
    def parla(self):  
        raise NotImplementedError("metodo astratto")
```

```
class Cane(Animale):  
    def __init__(self, nome):  
        self._nome = nome  
    def parla(self):  
        print("sono", self._nome, ", un cane ... bau")  
  
class Gatto(Animale):  
    def __init__(self, nome):  
        self._nome = nome  
    def parla(self):  
        print("sono", self._nome, ", un gatto ... miao")
```



<http://www.ce.unipr.it/brython/?p3 oop animals.py>

- **Actor**: interfaccia astratta
  - dichiara un metodo **move**, senza implementarlo
- vari **attori**: classi concrete
  - realizzano i metodi di Actor, definendo i **comportamenti specifici**
  - possono definire ulteriori metodi

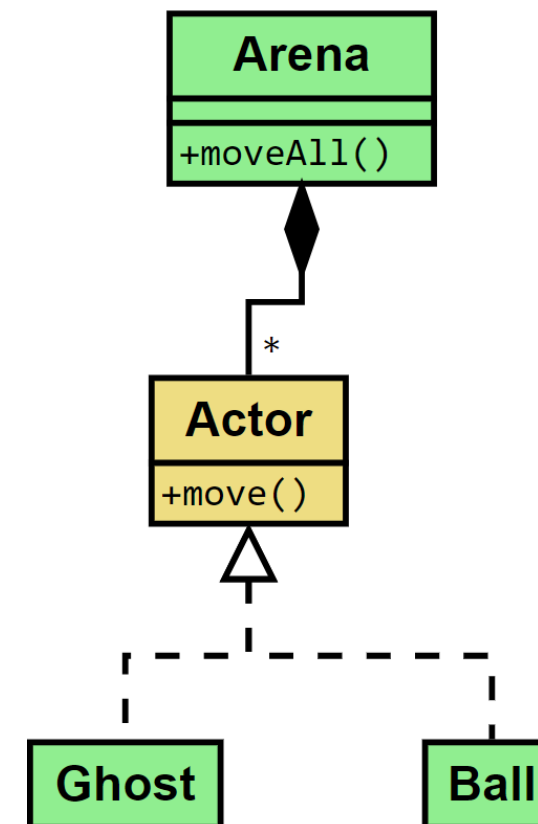
```
class Actor:  
    def move(self):  
        raise NotImplementedError("Abstract method")
```

- il codice è *dipendente* solo da (fa riferimento solo a) *interfacce astratte*
- nell'esempio *Arena* è riutilizzabile creando nuove classi concrete, che implementano *Actor*

```
class Arena: # ...
    def __init__(self, w, h):
        self._w, self._h = w, h
        self._actors = []
    def add(self, a: Actor):
        self._actors.append(a)
    def move_all(self):
        for a in self._actors:
            a.move()
    def size(self):
        return self._w, self._h
```



- principio di sostituzione di *Liskov*:
  - si può sempre *usare* un *oggetto* di una *classe derivata* *al posto* di uno della *classe base*
- relazione *has-a* tra un oggetto *Arena* e gli oggetti *Actor* che contiene
- relazione *is-a* tra classi concrete (*Ball* e *Ghost*) e interfaccia (*Actor*)



- metodo ***polimorfo***
- ***dichiarato*** in una interfaccia astratta
- ***implementato*** in *forme diverse* nelle classi concrete
- attori diversi possono muoversi in modo diverso

```
class Ghost(Actor): # ...
    def move(self):
        dx = random.choice([-5, 0, 5])
        dy = random.choice([-5, 0, 5])
        self._x = (self._x + dx) % ARENA_W
        self._y = (self._y + dy) % ARENA_H
```

```
class Ghost(Actor):  
  
    def __init__(self, arena, x, y):  
        self._x, self._y = x, y  
        self._arena = arena # save a ref to the arena  
        arena.add(self)      # register yourself into the arena  
  
    def move(self):  
        dx = random.choice([-5, 0, 5])  
        dy = random.choice([-5, 0, 5])  
        arena_w, arena_h = self._arena.size() # self._arena!  
        self._x = (self._x + dx) % arena_w  
        self._y = (self._y + dy) % arena_h
```

[http://www.ce.unipr.it/brython/?p3\\_bounce.py](http://www.ce.unipr.it/brython/?p3_bounce.py)

- al momento della **creazione** di un oggetto (istanziamento) viene **allocata memoria** per tenere lo stato dell'oggetto
- in **Python** una **variabile** è un **riferimento** ad un oggetto
- **oggetti**:
  - allocazione dinamica, in memoria heap
- **variabili**:
  - allocazione automatica, in memoria stack
- se un oggetto non è più associato a nessuna variabile: necessario liberare memoria
- **garbage collection**: gestione automatica della restituzione di memoria

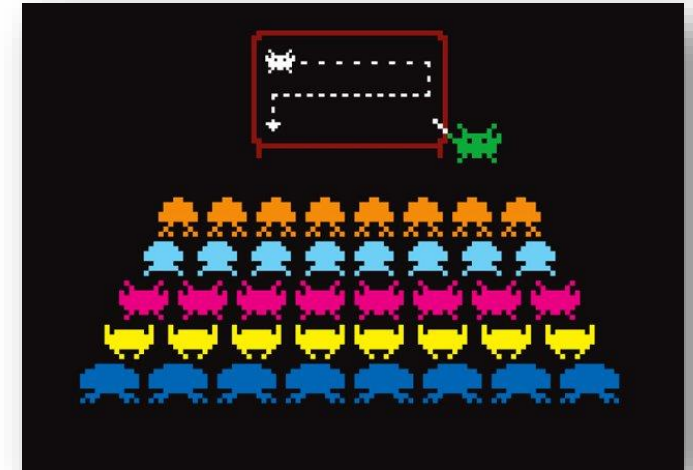
- ***vantaggi***
  - non è possibile dimenticare di liberare la memoria (***memory leak***)
  - non è possibile liberare della memoria che dovrà essere utilizzata in seguito (***dangling pointer***)
- ***svantaggi***
  - gc decide ***autonomamente*** quando liberare la memoria
  - liberare e compattare la memoria richiede del calcolo (operazione onerosa)
- esistono vari algoritmi di garbage collection



liste e tuple  
**esercizi**



- 6.1 alieni in arena
  - partire dalla classe Alien
  - renderla una implementazione concreta di Actor
  - aggiungere i metodi symbol e collide, anche solo vuoti
  - aggiungere gli alieni all'arena
  - nel metodo `__init__` chiamare `arena.add`
  - poi, in `update` chiamare `arena.move_all`



[http://www.ce.unipr.it/brython/?p3\\_oop\\_alien.py](http://www.ce.unipr.it/brython/?p3_oop_alien.py)