



# **Programovanie v jazyku Python**

OOP v Pythone, metametódy, vzťahy medzi triedami  
prednáška 9

Katedra kybernetiky a umelej inteligencie  
Technická univerzita v Košiciach  
Ing. Ján Magyar, PhD.

# Stav objektu

- stav objektu je súhrn hodnôt všetkých premenných objektu
- dva základné typy premenných:
  - členské premenné – jedinečné pre objekty
  - premenné triedy – jedinečné pre triedu
- stav objektu sa počas behu programu mení

# Premenné tried

- premenné typu `self.name` sú špecifické pre každú inštanciu
- Python umožňuje používanie premenných, ktoré zdieľajú všetky inštancie danej triedy
- definujeme ich mimo konštruktora
- najčastejšie aplikácie
  - jedinečný index (práca s databázami)
  - počítadlo
  - pomocná premenná pre niektoré návrhové vzory

# Viacnásobné dedenie v Pythone

```
class Person: pass
class Worker(Person): pass
class Boss(Person): pass

class Manager(Worker, Boss): pass
class Secretary(Worker): pass
class BoardMember(Boss): pass

class CEO(Manager, BoardMember): pass
```

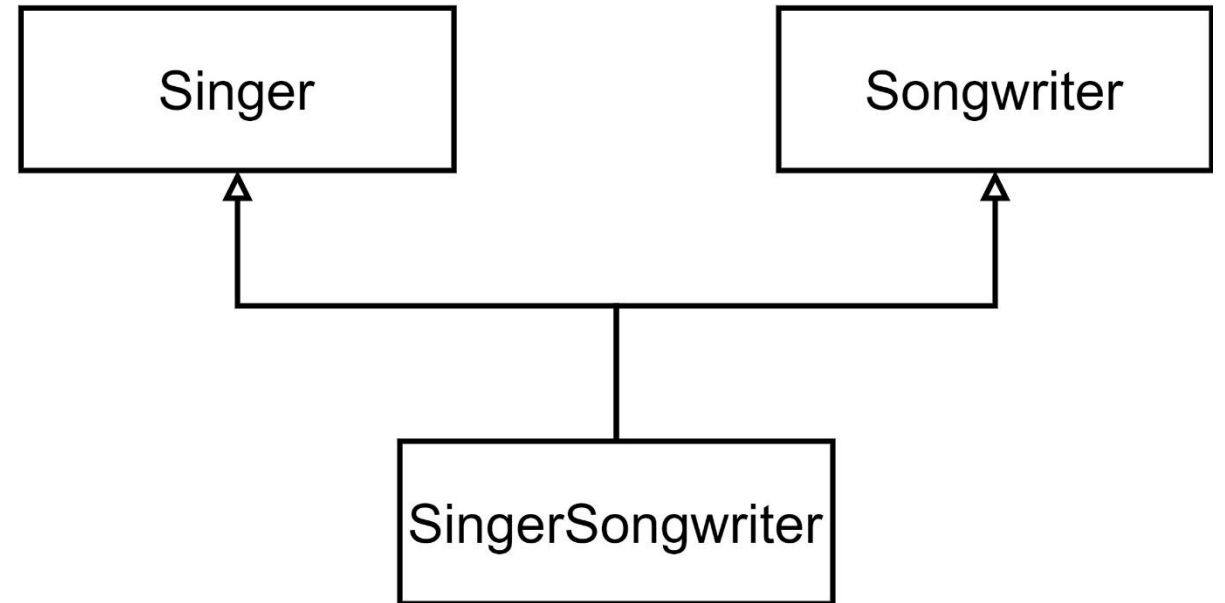
# Problém viacnásobného dedenia

v triede Singer:

```
sings = True  
writes = False
```

v triede Songwriter:

```
sings = False  
writes = True
```

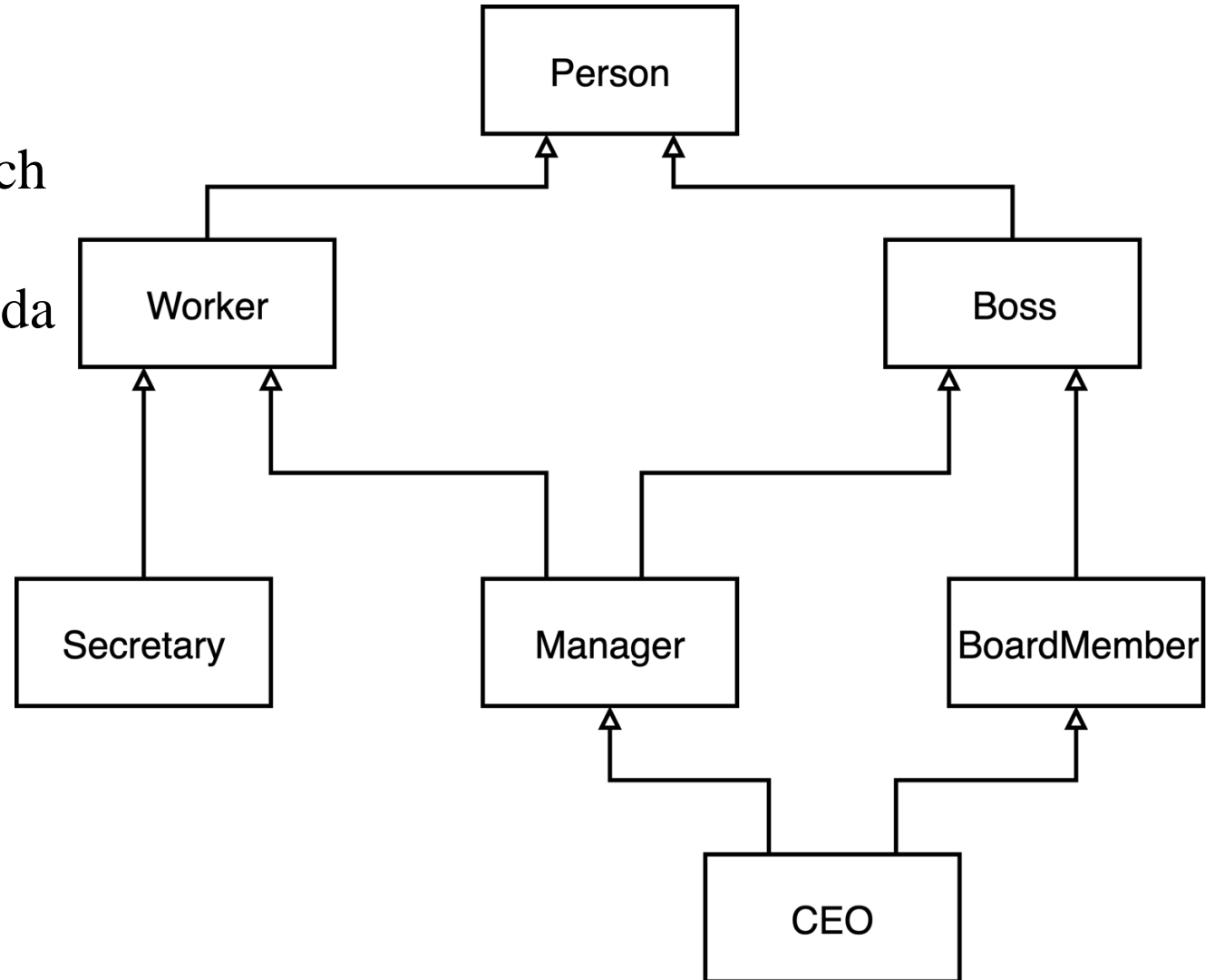


Aké hodnoty zdedí trieda  
SingerSongwriter?

# Name resolution pri viacnásobnom dedení

1. vyhľadávanie v aktuálnej triede
2. vyhľadávanie do hĺbky v nadtriedach a zľava doprava
3. ako posledná sa berie do úvahy trieda `object`

- poradie vieme zistiť pomocou premennej `__mro__` alebo metódou `mro()`



# Hlavné metametódy tried v Pythone

- `__init__`
- `__str__`
- `__eq__`, `__ne__`,  
`__lt__`, `__le__`,  
`__gt__`, `__ge__`
- `__hash__`
- `__del__`
- `dir()`

# Konštruktor

- metóda `__init__(self[, ...])`
- definuje spôsob vytvorenia inštancie triedy
- zavolá sa po spustení metódy `__new__` a vráti smerník na objekt
- konštruktor každej podtriedy musí zavolať konštruktor nadtriedy  
`SuperClassName.__init__(args)`



# Stringová reprezentácia objektu

- definuje sa v metóde `__str__(self)`
- použije sa pri volaniach
  - `str(my_object)`
  - `.format(my_object)`
  - `print(object)`
- má iba jednu návratovú hodnotu typu `string`

# Metódy rovnosti/nerovnosti

- slúžia na porovnávanie hodnôt premenných
- funkcionality v rôznych metódach

`__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, `__ge__`  
`==` `!=` `<` `<=` `>` `>=`

- návratová hodnota je zvyčajne `True` alebo `False`, ale môže byť ľubovoľná hodnota

# Metóda `__hash__` (`self`)

- používa sa pri volaní funkcie `hash()`, alebo pri operáciách s hašovanými skupinami hodnôt (`set`, `frozenset`, `dictionary` - kľúč musí byť hašovateľný)
- môže vracať ľubovoľnú hodnotu, jediná podmienka je že ak `object1 == object2`, tak `hash(object1) == hash(object2)`
- implementácia úzko súvisí s metódou `__eq__`: odporúča sa spojiť hodnoty, ktoré sa kontrolujú pri zisťovaní rovnosti dvoch objektov do jednej n-tice, a zavolať funkciu `hash()` nad touto n-ticou
- ak nemáte definovanú metódu `__eq__`, nemali by ste definovať ani `__hash__`; ak máte definovanú `__eq__` ale nie `__hash__`, trieda bude reprezentovať nehašovateľný typ

# Finalizer

- definovaný v metóde `__del__(self)`
- zavolá sa po volaní `del(my_object)`
- ak podtrieda definuje `__del__`, musí byť zavolaná metóda `__del__` nadtriedy
- odporúča sa použiť, ak objekt má aktívnu komunikáciu so súborom alebo s databázou – je potrebné uzavrieť tento komunikačný kanál
- v rámci `__del__` je možné vytvoriť nový smerník na aktuálny objekt - resurrection
- chyby, ktoré sa vyskytnú počas vykonávania metódy `__del__` sú ignorované, vypíše sa iba hláška na `sys.stderr`

# Funkcia `dir()`

- slúži na získanie všetkých atribútov daného objektu, resp. triedy
- pre objekt vráti zoznam vnútorných premenných, zoznam vnútorných metód, zoznam atribútov triedy do ktorej objekt patrí, a rekurzívne zoznam atribútov všetkých nadtried
- pre triedu vráti zoznam premenných triedy, zoznam metód triedy, a rekurzívne zoznam atribútov všetkých nadtried
- je možné definovať vlastný spôsob získania zoznamu atribútov v metóde `__dir__` ale zvyčajne to nie je potrebné

# Práca s objektmi ako s numerickými typmi

- je možné definovať spôsob, ako narábať s objektmi, ak sú argumentmi primitívnych algebraických operácií

|                       |   |
|-----------------------|---|
| +                     | <code>object.__add__(self, other)</code>      |
| -                     | <code>object.__sub__(self, other)</code>      |
| *                     | <code>object.__mul__(self, other)</code>      |
| /                     | <code>object.__truediv__(self, other)</code>  |
| //                    | <code>object.__floordiv__(self, other)</code> |
| %                     | <code>object.__mod__(self, other)</code>      |
| <code>divmod()</code> | <code>object.__divmod__(self, other)</code>   |
| **                    | <code>object.__pow__(self, other)</code>      |

# Práca s objektmi ako s numerickými typmi

|                              |  |
|------------------------------|--|
| <code>+=</code>              | <code>object.__iadd__(self, other)</code>      |
| <code>-=</code>              | <code>object.__isub__(self, other)</code>      |
| <code>*=</code>              | <code>object.__imul__(self, other)</code>      |
| <code>/=</code>              | <code>object.__itruediv__(self, other)</code>  |
| <code>//=</code>             | <code>object.__ifloordiv__(self, other)</code> |
| <code>%=</code>              | <code>object.__imod__(self, other)</code>      |
| <code>**=</code>             | <code>object.__ipow__(self, other)</code>      |
| <code>abs()</code>           | <code>object.__abs__(self)</code>              |
| <code>complex(object)</code> | <code>object.__complex__(self)</code>          |
| <code>int(object)</code>     | <code>object.__int__(self)</code>              |
| <code>float(object)</code>   | <code>object.__float__(self)</code>            |

# Zaokrúhľovanie hodnoty objektu na celé čísla

|                      |  |
|----------------------|--|
| <code>round()</code> | <code>object.__round__(self[, ndigits])</code> |
| <code>trunc()</code> | <code>object.__trunc__(self)</code>            |
| <code>floor()</code> | <code>object.__floor__(self)</code>            |
| <code>ceil()</code>  | <code>object.__ceil__(self)</code>             |



# Ukážka - hierarchia tried

- vytvoríme hierarchiu nadtried a podtried
- dedenie a prepisovanie metód
- práca s podtriedami
- iterátory

# Prepisovanie metód v Pythone

- nazýva sa to aj shadowing - prekonávanie
- špecifikujeme inú funkcionálnosť pre podtriedy, ale použijeme rovnaký názov metódy aj rovnaké parametre
- ak funkcionality nie sú úplne iné (nemali by byť), tak by sme mali využiť implementáciu z nadtriedy

# Pret'aženie metód v Pythone

- máme metódy s rovnakým názvom, ale s inou návratovou hodnotou/s inými parametrami
- typický príklad – pret'aženie základných operácií
- v Pythone pret'aženie vlastných metód nie je možné, používajú sa na rovnaký účel defaultné hodnoty

## Čo by sme chceli:

```
class Number:
    def __init__(self, value):
        self.num = value

    def multiply(self):
        return self.num * 2

    def multiply(self, number):
        return self.num * number
```

## Ako to urobíme:

```
class Number:
    def __init__(self, value):
        self.num = value

    def multiply(self, number=2):
        return self.num * number
```

## Čo by sme chceli:

```
class Number:
    def __init__(self, value):
        self.num = value

    def multiply(self):
        return self.num * self.num

    def multiply(self, number):
        return self.num * number
```

## Ako to urobíme:

```
class Number:
    def __init__(self, value):
        self.num = value

    def multiply(self, number=None):
        if number is None:
            return self.num * self.num
        else:
            return self.num * number
```

alebo použijeme iný názov metódy

# Práca s podtriedami

- keďže podtriedy by mali byť bližšie špecifikácie nadtriedy, funkcionality preťažených tried by mala byť podobná
- práve preto je dobrým zvykom použiť implementáciu z nadtriedy (volanie `super`)
- v Pythone máme dve možnosti:
  - ak poznáme názov nadtriedy (zvyčajne):  
`SuperClass.method_name(self, parameters)`
  - ak nepoznáme názov nadtriedy (chceme vytvoriť podtriedu z knižnice):  
`super(SubClass, self).method_name(parameters)`

# Iterátory

- pre kolekcie
- definujú podporu pre `for` cykly
- definícia pomocou dvoch metód
  - `__iter__(self)`
    - vytvorí iterátor
    - inicializuje pomocné premenné
    - vracia `self`
  - `__next__(self)`
    - vracia nasledujúci prvok v kolekcii
    - ak sme sa dostali na koniec kolekcie, vyhodí výnimku `StopIteration`

# UML diagramy

- UML – Unified Modeling Language
- slúži na vizualizáciu architektúry a funkcionality softvérového riešenia
- základom objektovo orientovaného modelovania je diagram tried (class diagram)
- jeden blok reprezentuje jednu triedu
- môže byť použitý aj pre modelovanie dát



# Diagram triedy

|              |
|--------------|
| názov triedy |
| atribúty     |
| metódy       |

|  |
|--|
| Guitarist  |
| family_name: string<br>first_name: string<br>idNum: int<br><u>nextIdNum: int</u>   |
| __init__(familyName: string, firstName: string)<br>getIdNum(): int<br>__str__(): string<br>__eq__(other: Guitarist): boolean |

# Enkapsulácia

- v Pythone nemáme príznaky, definujeme to priamo v názve premennej
  - `idNum: int` - public premenná
  - `__idNum: int` - private premenná
- premenné triedy sú podčiarknuté
  - `nextIdNum: int`

# Vzt'ahy medzi triedami

- asociácia
- dedenie
- implementácia
- závislosť
- agregácia
- kompozícia



Association



Inheritance



Realization /  
Implementation



Dependency



Aggregation



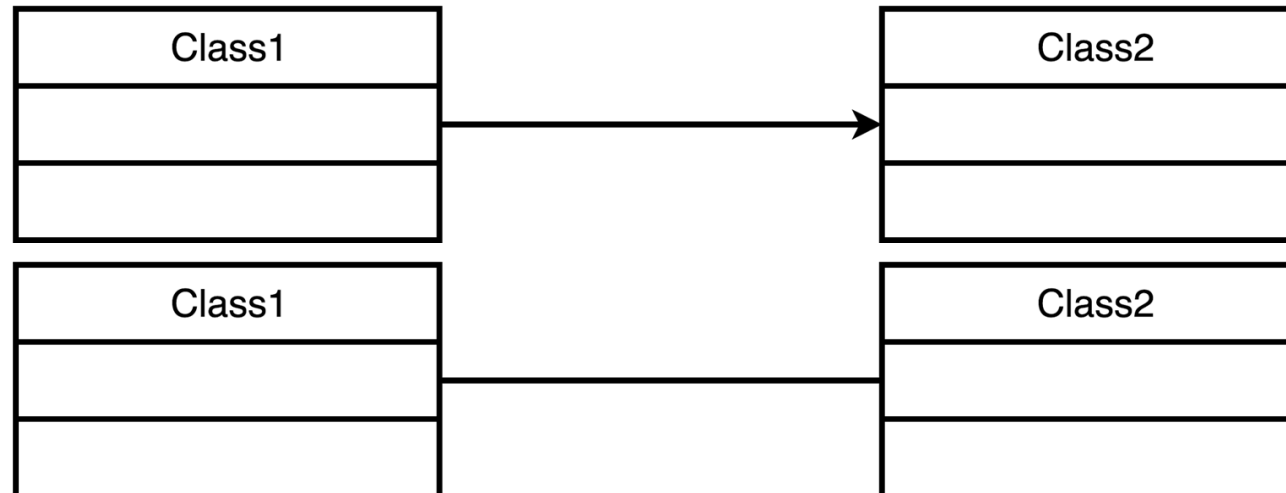
Composition

# Kardinalita

- pre každý vzťah môžeme definovať multiplicitu (koľkonásobný je vzťah)
  - **0** - 0
  - **0..1** - 0 alebo 1
  - **0..\*/\*** - 0 až n
  - **1/1..1** - 1
  - **1..\*** - 1 až n
- uvádza sa na konci čiary reprezentujúcej vzťah (pri triede)

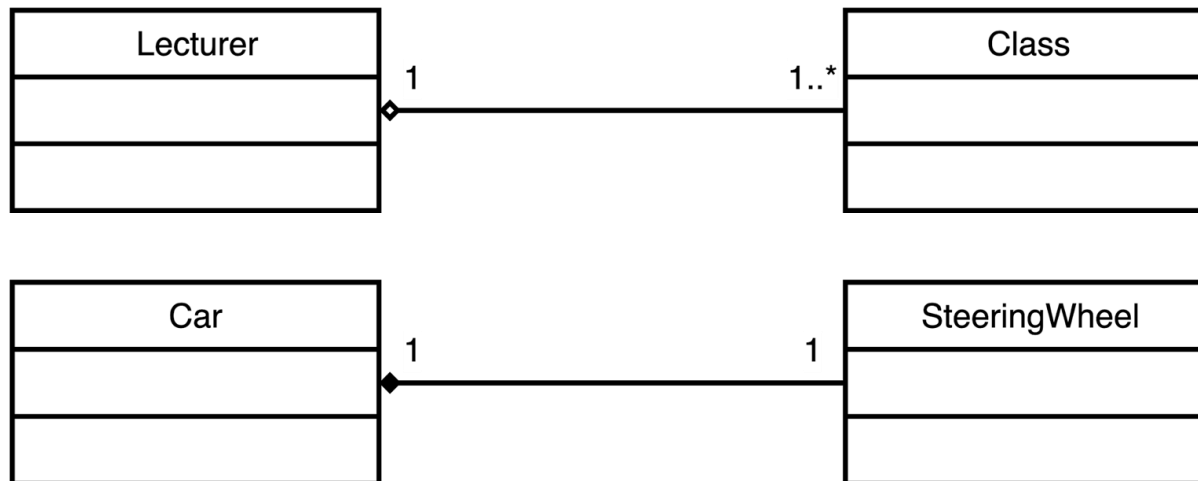
# Asociácia

- na úrovni inštancií
- reprezentovaná šípkou (jednosmerná asociácia) alebo čiarou (obojsmerná)
- objekt jednej triedy sa spolieha na metódu druhého objektu
- jeden objekt používa druhý objekt
- jeden objekt je atribútom druhého objektu



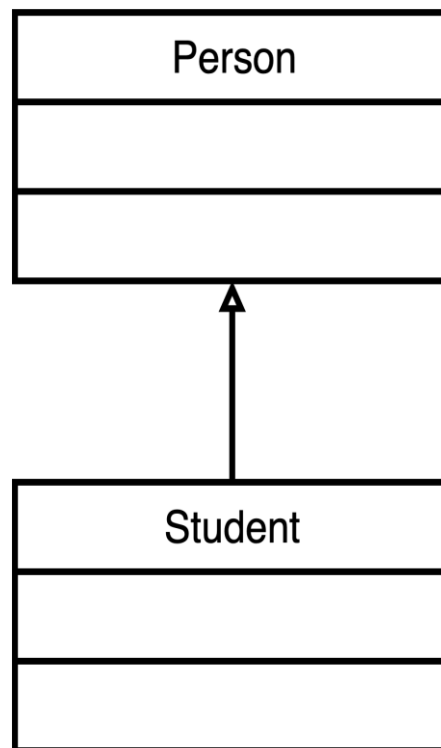
# Agregácia a kompozícia

- na úrovni inštancií
- vyjadrujú vzťah, kde objekt jednej triedy sa skladá/obsahuje objekt druhej triedy
- agregácia – ak vymažeme kontajner objekt, jednotlivé časti môžu ďalej existovať
- kompozícia – jednotlivé časti nemajú funkcionality mimo kontajnera



# Dedenie

- na úrovni tried
- dedenie vyjadruje, že podtrieda je bližšou špecifikáciou nadtriedy



# Závislosť

- všeobecný vzťah
- vyjadruje prípad, kde jeden objekt použije druhý objekt, ale vzťah je omnoho slabší ako asociácia
- trieda, ktorá je závislá od druhej, obsahuje metódu, kde objekt z nezávislej triedy je parameter metódy





# Zhrnutie

- stav objektu
- viacnásobné dedenie
- členské premenné a premenné tried
- konštruktor
- metametódy a ich význam
- porovnávanie objektov
- prepisovanie a preťaženie
- UML diagramy, vzťahy medzi triedami