07-02-2026

Agenda:

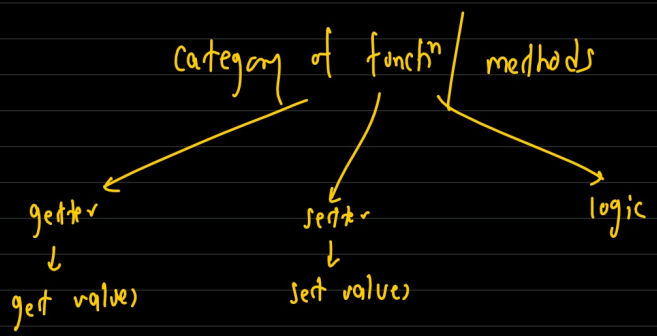
- OOP - II
- modules

OOP - II

→ class → Blueprint ⟷ constructor → common connection → self
→ object → instance

4 pillars of OOP:

* Encapsulation → (Data hiding) (Scope)
* Abstraction → (Contract) (Hiding complex implementation)
* Inheritance → (code reuse)
* Polymorphism (Many forms) (ability of different object to respond in a similar way)



Bank → deposit → setter
→ view balance → getter

Scope levels in python

```
class BankAccount:
    def __init__ (self, name, balance):
        self.name = name # public
        self._account_type = "Savings" # Protected
        self.__balance = balance # private

    def get_balance(self):
        # getter()
        # to get value of the balance variable
        print("Your current balance :", self.__balance)

    def add_and_set_new_balance(self, amount):
        # setter()
        # to deposit money in our account
        self.__balance += amount
        print("Your current balance :", self.__balance)
```

obj = BankAccount("a", 100)

obj.getbalance() → 100

obj.__balance → error

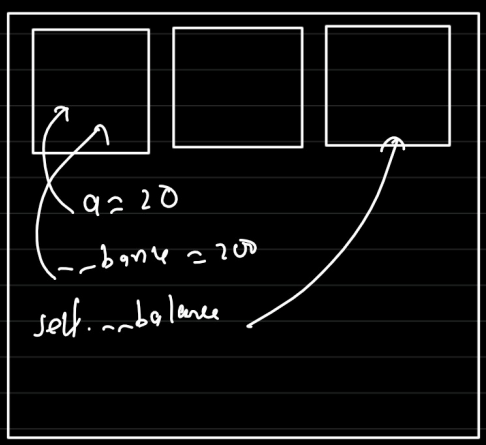
obj.__balance = 200
↳ create a new variable
= 200

private variable → python renames to → _ClassName__private_variable_name (name mangling)

class Hello:

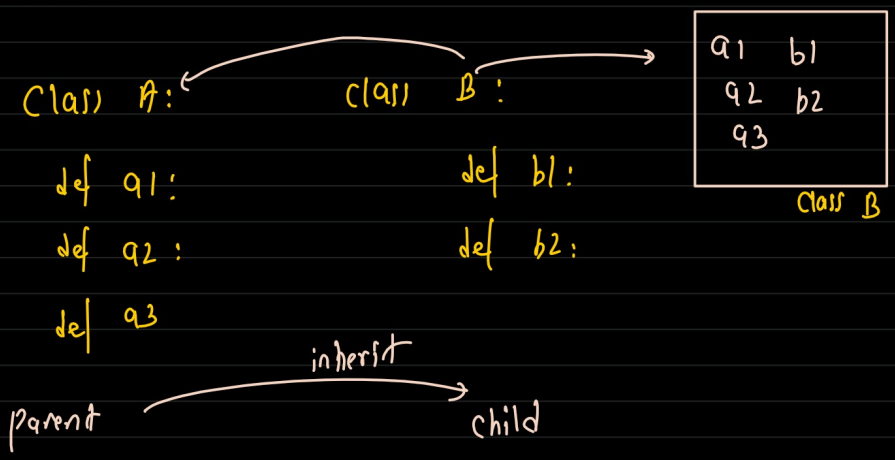
self.__a # private → rename → _Hello__a

Application → class name

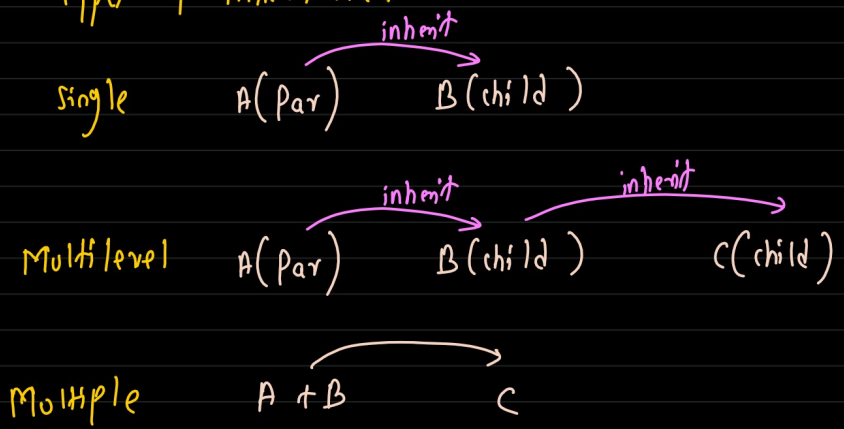


Encapsulation : Scope + control

Inheritance:



Type of inheritance:




```
class Device :
    def power_on(self):
        print("Device On")

class Tv(Device):
    def play_channel(self):
        print("play channel")
```

inheritance :—

IS-A rule

IS-A relationship

Dog IS-A Animal

```
class Animal :
    def speak :
        pass

class Dog :
    def bark:
        pal
```

Car HAS-A Engine

```
class car :
    def wheel()
    def drive()
    e = Engine()

class Engine :
```

Design concept (optional)

→ a = [1, 2, 3]

→ a[0] = 2

→ [2, 2, 3]

```
class A(B):
    def a(self):
        ....()
```

```
obj1 = A()
```

→ address → class A

self

```
obj1.a()
```

self → class A - a

```
class B :
    def __init__(self):
        self.name = "Data"

    def b(self):

    def a(self):
```

class B.a

super() →

* do we have parent class → yes → B()

* B().a() →

class MotherBoard :

def __init__(self)

→ ram
→ cpu_speed = ghz
→ time =
→ cpu_type =

def start():

def stop()

class computer(MotherBoard):

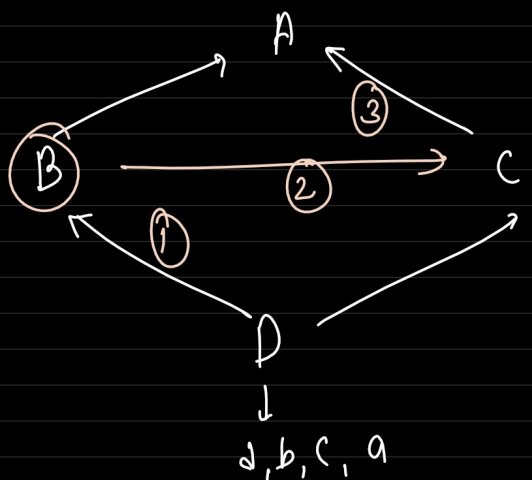
def __init__(self):

def start computer()

if cpu_type = "apple"

start macos

def start :



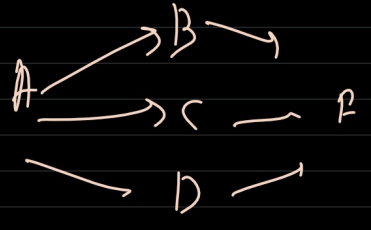
d, b, c, a

MRO

→ Method Resolution order

↓

used to find next method in the inheritance chain

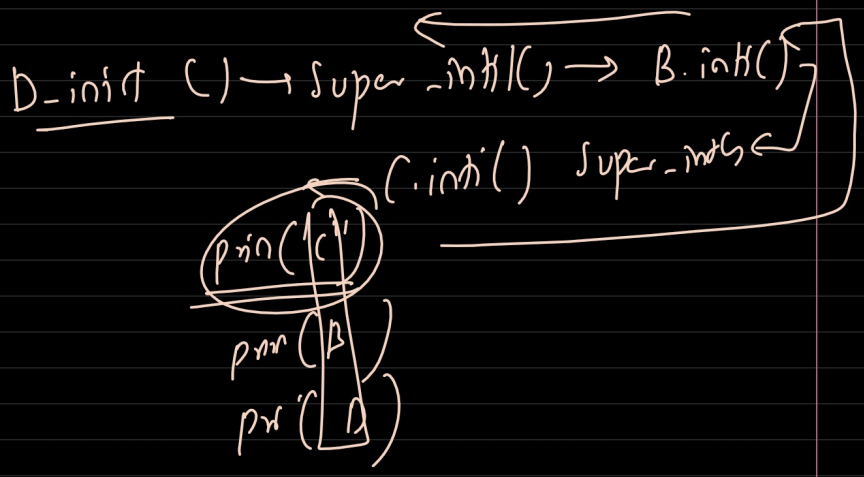


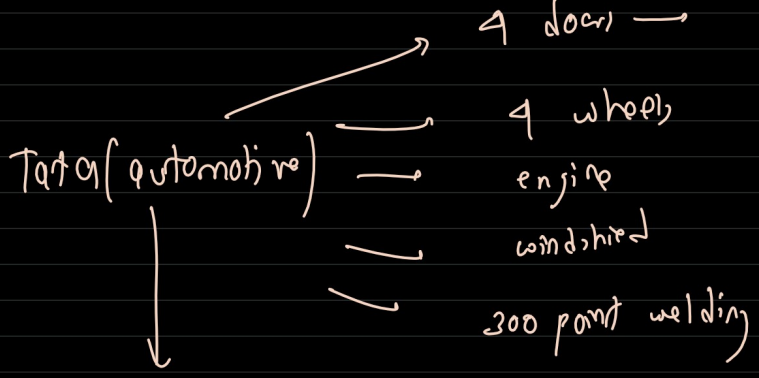
```python
class A:
    def __init__(self):
        print("Class A")
    def a(self):
        print("a")

class B(A):
    def __init__(self):
        super().__init__()
        print("Class B")
    def b(self):
        print("b")

class C(A):
    def __init__(self):
        print("Class C")
    def c(self):
        print("c")

class D(B,C):
    def __init__(self):
        super().__init__()
        print("Class D")
    def d(self):
        print("d")
```

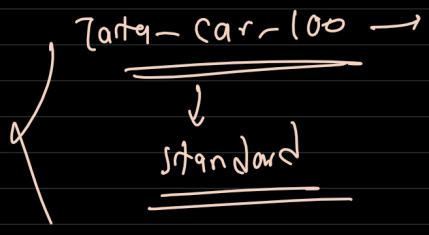
```python
d_obj = D()
```

```
Class C
Class B
Class D
```

```
# D --> B --> C --> A
```



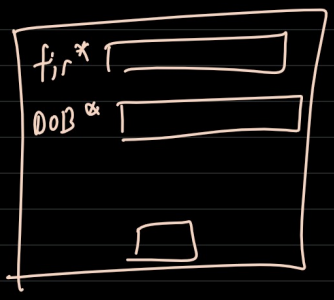


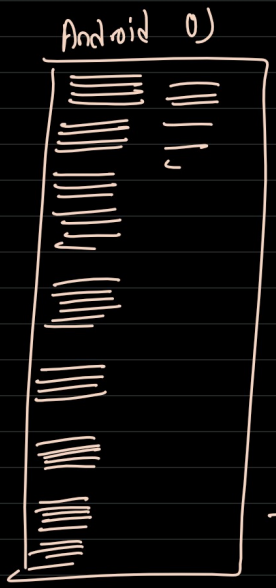


Abstract class / interface like / contract

from abc, import ABC, abstractmethod

Online → pay
Bank ← refund





→ 10,00000 — 1 million line of code

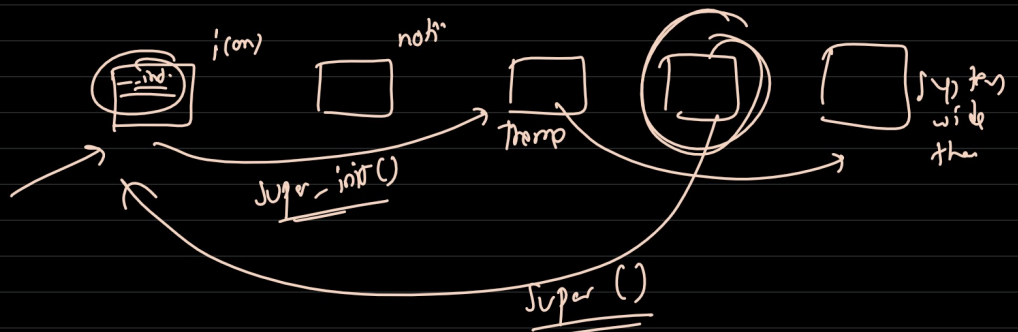
— Android OJ
→ to fix the icon and adjust the box with

2 bit

→ procedural programming

object → OOP!

1.5 million



Python → Encapsulation → Security

↓

design →

hello	hello()
_hello	_hello()
__hello	__hello()

inheritance →

Abstract → Bank → contract for customer ank

Polymorph