Object Oriented Programming (OPP):

1- Encapsulation.

2- Polymorphism.

3- Abstraction.

4- Inheritance.

Primitive Data Types: saves single simple Value, like Byte int float Boolean Char

Primitive data types were not a problem for simple programs but when the programs started to get complex, we had to make grouping for variables together, example: we want to programming a chess game, the chess pieces ( as king, Queen, Bishop, knight, rook, pawn) need many variables like position, color(white or black), Boolean(1:dead, 0 live).., in array we can save more variables but with the same type, we need a new thing that can save more variable types structure can save more variable type but we have with structure the problem that we cant define function inside the structure ( we need in our chess game functions like move() that allow us to move inside the checkerboard🡪 so we need objects

An object is an instance of a class and the class is a template for the object.

Class is bishop(with color-var, position-var, color-var, and move()-function) but in class we don’t add values because class is general for all bishops but we can define the move()-function because all bishops have the same movement. The object of the class bishop had a new name like b\_1→

b\_1=new bishop(3, “white”, false);

-------------------------------------------------------------------------------------------------

1- Encapsulation (Black Box):

We have to handle classes with fun functions like (Setter & Getter) so Encapsulation means we cant handle Objects variable without these Setter and Getter functions because if we handle objects without Setters we can man mistakes like move bishop like a rook. And with Setter we can make other functions with out command example if we move bishop to other place where there is another bishop we can make the other bishop automatically dead.

So the concept of encapsulation: The access to Object should be just with Setters and Getters so we cannot allow the impossible movement.

----------------------------------------------------------------------------------------------

2-Abstraction: show just important details. If many people work on project we can with abstraction divide the project. That I know what make this class without to know how. I need to know if I have Check Mate or not with function check\_mate() but how this function works is not important.

The idea of Libraries is abstraction

The methode like chek\_mate() called Interface.

-----------------------------------------------------------------------------------------------

3- Inheritance (parent & child):

Example: Student-Calss and Teacher-Class have similar properties(name, address ), and differences ( with student ( degree) with teacher ( salary) so we make Parent class Person-Class with Attributes( name and address) and two Child-Classes (student(with degree and inheritance of person-class) and teacher (with salary and inheritance of person-class).

----------------------------------------------------------------------------------------------

4-Polymorphism (poly: has many and morphism: forms):

Animal-class with methode makeSound() and Dog-class with the same methode() and cat-class with the same methode makeSound() and with cat and dog I should make overwriting of makeSound() with the suitable Sound.

Method: is function inside a class

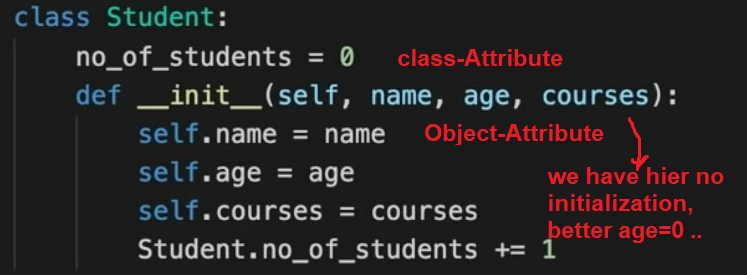
-----------------------------------------------------------------------------------------------

Practice:

Instance=Object (instance of a class = object of a class)

Attributes are properties of class (with student attributes are name age number …)

Class-attribute Vs. Object-Attribute:



**In instance-method (objects-method) we always need the self-parameter**.

**In class-method we always need the cls-parameter**.

-very helpful method: with f “”

def describe(self):  
 print(f"my name is {self.name} and my age is {self.age}")

or with .format()

def describe(self):  
 print("my name is {} and my age is {}".format(self.name,self.age))

we should access attributes with setter() but how can we forbid the command student\_1.name=..

→ we make the attributes private with the command in \_\_init\_\_ : self.\_\_name=..

Wenn we now make student\_1.name=”x” then we create an new attribute for the object student\_1 but our attribute name can not be accessed now🡪 name ≠ \_\_name

🡪🡪 This is the encapsulation concept (no direct access).

**Class-Method Vs. Instance-Method:**

from datetime import date  
class Student:  
 def \_\_init\_\_(self,name,age):  
 self.name = name  
 self.age = age  
  
 def desc(self): # Instance-/Object-Method  
 print("my name is {} and my age is {}".format(self.name,self.age))  
  
 @classmethod # Class-Method  
 def initfrombirthyear(cls,name,birthYear):  
 return cls(name,date.today().year - birthYear)  
  
student\_1=Student("Mohammad",29)  
student\_2=Student.initfrombirthyear("Ahmad",1991)  
student\_1.desc()  
student\_2.desc()

Output:

my name is Mohammad and my age is 29

my name is Ahmad and my age is 31

-----------------------------------------------------------------

Other Example:

class Pizza:  
 def \_\_init\_\_(self, ingredients):  
 self.ingredients=ingredients  
  
 @classmethod  
 def veg(cls):  
 return cls(["mushrooms", "olives","onions"])  
  
 @classmethod  
 def margherita(cls):  
 return cls(["mozarella", "suade"])  
  
pizza\_1=Pizza(["tomatos","olives"])  
pizza\_2=Pizza.veg()  
pizza\_3=Pizza.margherita()  
print(pizza\_1.ingredients)  
print(pizza\_2.ingredients)  
print(pizza\_3.ingredients)

Outout:

['tomatos', 'olives']

['mushrooms', 'olives', 'onions']

['mozarella', 'suade']

Other way to write this program with handle the standard function \_\_str\_\_

class Pizza:  
 def \_\_init\_\_(self, ingredients):  
 self.ingredients=ingredients  
  
 @classmethod  
 def veg(cls):  
 return cls(["mushrooms", "olives","onions"])  
  
 @classmethod  
 def margherita(cls):  
 return cls(["mozarella", "suade"])  
  
pizza\_1=Pizza(["tomatos","olives"])  
pizza\_2=Pizza.veg()  
pizza\_3=Pizza.margherita()  
  
print(pizza\_1,pizza\_2,pizza\_3)

Output:

<\_\_main\_\_.Pizza object at 0x000001EF448EDF10>

<\_\_main\_\_.Pizza object at 0x000001EF448EDD90>

<\_\_main\_\_.Pizza object at 0x000001EF448EDD30>

The output is the standard \_\_str\_\_ so we need to edit this standard function:

def \_\_str\_\_(self):  
 return f"Pizza ingredients are {self.ingredients}.\n"

this full program looks like this:

class Pizza:  
 def \_\_init\_\_(self, ingredients):  
 self.ingredients=ingredients  
  
 @classmethod  
 def veg(cls):  
 return cls(["mushrooms", "olives","onions"])  
  
 @classmethod  
 def margherita(cls):  
 return cls(["mozarella", "suade"])  
  
 def \_\_str\_\_(self):  
 return f"Pizza ingredients are {self.ingredients}.\n"  
  
pizza\_1=Pizza(["tomatos","olives"])  
pizza\_2=Pizza.veg()  
pizza\_3=Pizza.margherita()  
  
print(pizza\_1,pizza\_2,pizza\_3)

The Output:

Pizza ingredients are ['tomatos', 'olives'].

Pizza ingredients are ['mushrooms', 'olives', 'onions'].

Pizza ingredients are ['mozarella', 'suade'].

-----------------------------------------

The @classmethod above called decorator.

Know static method with the decorator @staticmethod +dunder functions + Abstraction → https://www.youtube.com/watch?v=A9kSngn7254&t=7215s

----------------------------

Functions with two underscores called dunder-functions like \_\_init\_\_ and the are standard but we can overwrite them and to show them we write dir() and in the brackets the name of the Class

→ print( dir ( Class-Name ) ) ;

--------------------------------------------

ASoleman