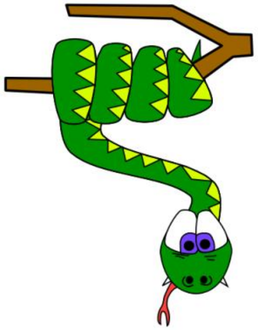


Object Oriented Programming

```
class stud:
def set(self,nm,ag):
    self.name=nm      #public: No underscore
    self.age=ag
def setstream(self,st):
    self.__stream=st  #private : double underscore
def setclass(self,cl):
    self._cl=cl       #protected: single underscore
def getstream(self):
    return self.__stream
def getName(self):
    return self.name
def getage(self):
    return self.age
d=stud()
d.set("ram",32)   # name and age passed to set method
print(d.getName(),d.getage())
print(d.name,d.age)
d.setstream("science")
d.setclass(9)
print(d._cl)
print(d.getstream())
print(d._stud__stream)#name mangling to access private data
```
Dr Harsh Dev

```
class demo:
    x=5
    def disp(self,x):
        x=30
        #local variable tends to hide the value of the instance variable
        self.x=10
        print('local',x)
        print('instance',self.x)
d=demo()
d.disp(50)
print(dir(demo))
print(d.x)
print(demo.x)
print(demo.__dict__)
print(d.__dict__)
```

Class Method & Constructor

```
class customer:
    counter=1000
    def __init__(self):
        customer.counter=customer.counter+1
        self.cid=customer.counter
    def setAttr(self,nm,ag):
        self.name=nm
        self.age=ag
    @classmethod
    def total_customer(cls):
        return customer.counter-1000
c1=customer()
c1.setAttr("ram", 15)
c2=customer()
c2.setAttr("shyam",36)
print(customer.total_customer())
```

Multiple Constructor

```
class demo:
    def __init__(self,*var):   #constructor
        if len(var)==2:
            self.name,self.age=var  # var is a tuple
        elif len(var)==1:
            self.name=var
        else:
            self.name=input("Enter name")
            self.age=eval(input("Enter age"))
d=demo("ram",20)
d1=demo("shyam")
d2=demo()
print(d.name,d.age)
print(d1.name)
print(d2.name,d2.age)
```

Dr Harsh Dev

Destructor method __del__()

- Like other oop languages Python also has a destructor.
- the method __del__() denotes the destructor and the syntax to define the destructor is:
- def __del__(self):

  block of statements

Python invokes the destructor method when the instance is about to be destroyed. It is invoked one per instance.

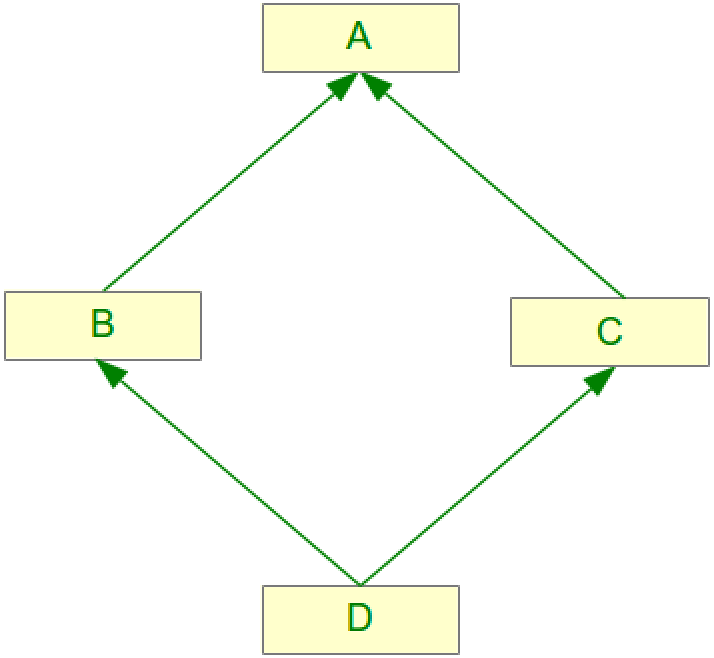
The self refers to the instance on which the __del__(self) method is invoked.

Python manages garbage collection of objects by reference counting. This function is executed only if all the references to an object have been removed.

```
class DestructDemo:
    def __init__(self):
        print("you are welcome")
    def __del__(self):
        print("dectructor executed successfully")
ob1= DestructDemo()
ob2=ob1
ob3=ob1
print("id of ob1 is: ",id(ob1))
print("id of ob2 is: ",id(ob2))
print("id of ob3 is: ",id(ob3))
del ob2 #reference count becomes 2
del ob1 #reference count becomes 1
del ob3 #reference count becomes 0
#destructor called automatically since reference count becomes zero
a=input("input string")
print(a)
```

The Diamond Problem or the "deadly diamond of death"

- class **A():**
- def **m(self):**
- print("method m of A called")
- class **B(A):**
- def **m(self):**
- print("method m of B called")
- class **C(A):**
- def **m(self):**
- print("method m of C called")
- class **D(B,C):**
- pass



- diamond = D()
- **#Method resolution order: bottom up and left to right**
- diamond.m()

```python
class A:
    def __init__(self):
        print("A.__init__")
class B(A):
    def __init__(self):
        print("B.__init__")
        super().__init__()
 class C(A):
    def __init__(self):
        print("C.__init__")
        super().__init__()
class D(C,B):
    def __init__(self):
        print("D.__init__")
        super().__init__()
d = D()
print(D.mro())
```

- **It uses the so-called method resolution order(MRO).**
- **It is based on the "C3 superclass linearization" algorithm.**
- **This is called a linearization, because the tree structure is broken down into a linear order.**
- **The mro() method returns list of method resolution order**

Method overloading in Python

- class OverLoad:
-     def add(self,a,b):
-         print(a+b)
-     def add(self,a,b,c):
-         print(a+b+c)
- p=OverLoad()
- p.add(10,20)

- #**Produces error** because Python understands the last definition of the method add(self,a,b,c) apart from the self
- #Python does not allow method overloading based on type as it is strongly typed language

- class **methodoverloademo:**
-     def **add(*self,*var*):*#constructor***
-         if len(var)==2:
-             a,b=var
-             return a+b
-         if len(var)==3:
-             a,b,c=var
-             return a+b+c
-
- m1=methodoverloademo()
- m2=methodoverloademo()
- print(m1.add(2,3))
- print(m2.add(2,3,4))

Operator Overloading

- A programmer can overload almost every operator, such as arithmetic, comparison, indexing ans slicing and the number of inbuilt functions.
- To support operator overloading Python associates a special method with each inbuilt function and operator.
- If we have an expression "x + y" and x is an instance of class K,
- then Python will check the class definition of K. If K has a method __add__ it will be called with x.__add__(y), otherwise we will get an error message.

Magic Methods and Operator Overloading

- magic methods are sometimes called dunder methods!
  So what's magic about the __init__ method? The answer is, you don't have to invoke it directly. The invocation is realized behind the scenes. When you create an instance x of a class A with the statement "x = A()", Python will do the necessary calls to __new__ and __init__.
  It's even possible to overload the "+" operator as well as all the other operators for the purposes of your own class. To do this, you need to understand the underlying mechanism. There is a special (or a "magic") method for every operator sign. The magic method for the "+" sign is the __add__ method. For "-" it is "__sub__" and so on. We have a complete listing of all the magic methods a little bit further down.
  The mechanism works like this: If we have an expression "x + y" and x is an instance of class K, then Python will check the class definition of K. If K has a method __add__ it will be called with x.__add__(y), otherwise we will get an error message.

- **Binary Operators**
- Operator Method

Operator	Method
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__truediv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

- class **OperatorOverLoad:**
-     def **__init__(*self,x*):**
-         *self.x =x*
-     def **__add__(*self,other*):**
-         print(*'the value of ob1=',self.x)*
-         print(*'the value of ob2=',other.x)*
-         print(*'The addition of two objects is: ')*
-         return *self.x+other.x*
- ob1=OperatorOverLoad(20)
- ob2=OperatorOverLoad(30)
- print(ob1+ob2)   **#ob1.__add__(ob2)**

- **Extended Assignments**
- **Operator** **Method**

Operator	Method
+=	object.__iadd__(self, other)
-=	object.__isub__(self, other)
*=	object.__imul__(self, other)
/=	object.__idiv__(self, other)
//=	object.__ifloordiv__(self, other)
%=	object.__imod__(self, other)
**=	object.__ipow__(self, other[, modulo])
<<=	object.__ilshift__(self, other)
>>=	object.__irshift__(self, other)
&=	object.__iand__(self, other)
^=	object.__ixor__(self, other)
|=	object.__ior__(self, other)

- **Unary Operators**

Operator	Method
-	object.__neg__(self)
+	object.__pos__(self)
abs()	object.__abs__(self)
~	object.__invert__(self)
complex()	object.__complex__(self)
int()	object.__int__(self)
long()	object.__long__(self)
float()	object.__float__(self)
oct()	object.__oct__(self)
hex()	object.__hex__(self

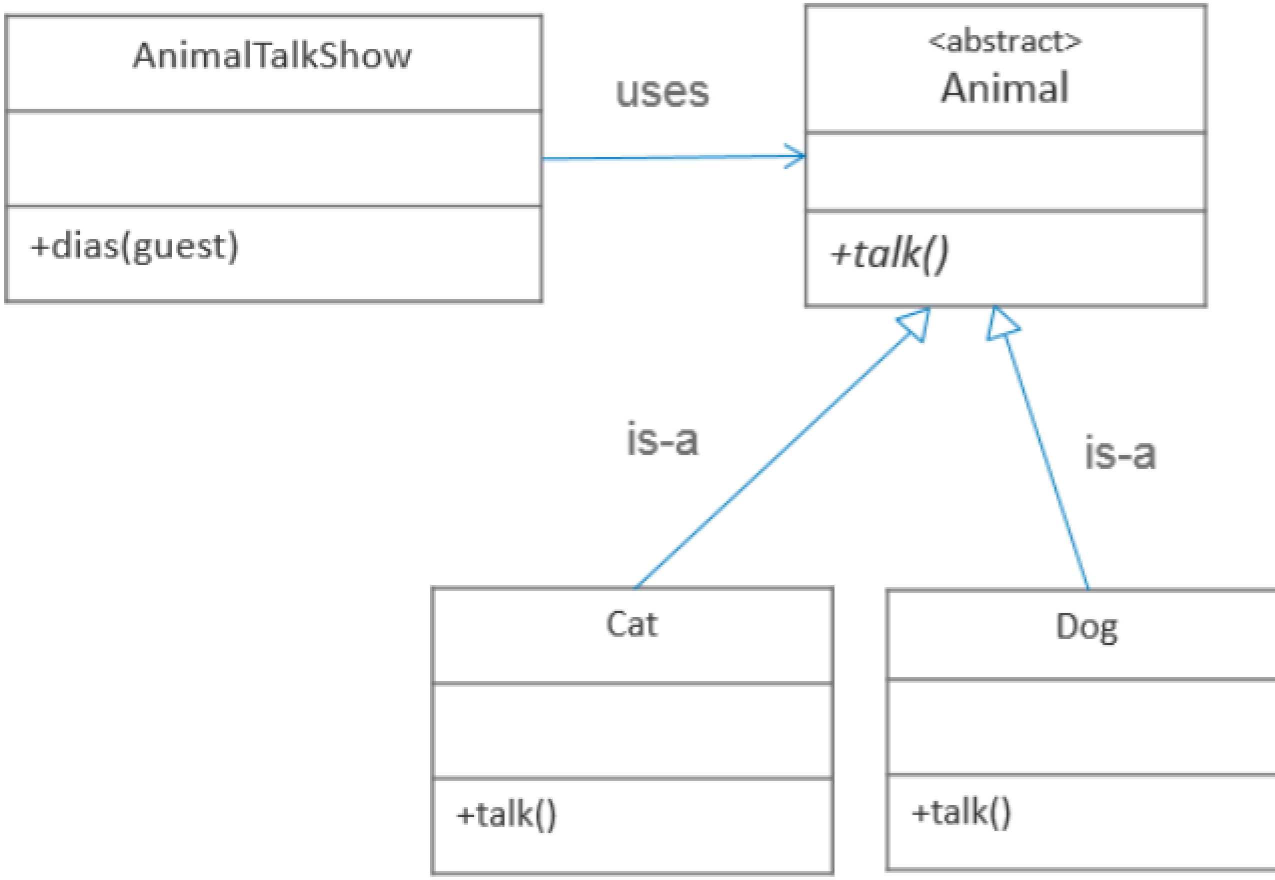
```python
from math import sqrt
class OperatorOverLoad:
    def __init__(self,x,y):
        self.x =x
        self.y=y
    def __eq__(self,other):
        print('the value of ob1=',str(self.x)+'+'+str(sqrt(2))+'*'+str(self.y))
        print('the value of ob2=',str(other.x)+'+'+str(sqrt(2))+'*'+str(other.y))
        print('equality of two objects is: ')
        if str(self.x)+'+'+str(sqrt(2))+'*'+str(self.y)==str(other.x)+'+'+
                                                            str(sqrt(2))+'*'+str(other.y):
            return True
        #return str(self.x+other.x)+'+'+str(sqrt(2))+'*'+str(self.y+other.y)
ob1=OperatorOverLoad(20,30)
ob2=OperatorOverLoad(20,30)
print(ob1==ob2)    #ob1.__add__(ob2)
```

- **Comparison Operators**
- Operator Method
  - < object.__lt__(self, other)
  - <= object.__le__(self, other)
  - == object.__eq__(self, other)
  - != object.__ne__(self, other)
  - >= object.__ge__(self, other)
  - > object.__gt__(self, other)

Cats and Dogs participated in an Animal talk show.

Only Animals can go to talk show dias and talk.

- A cat talks "Meow !"
- A dog talks "Woof !"

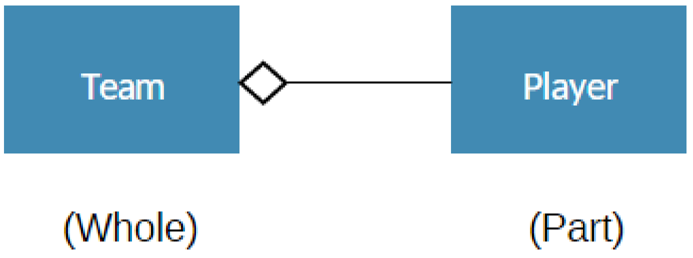


```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def talk(self):
        pass

class Cat(Animal):
    def talk(self):
        print("Meow !")

class Dog(Animal):
    def talk(self):
        print("Woof !")

class AnimalTalkShow:
    def dias(self,guest):
        #Ensure that the guest is a valid animal
        if (isinstance(guest, Animal)):
            #actual behavior is selected dynamically
            guest.talk()
```

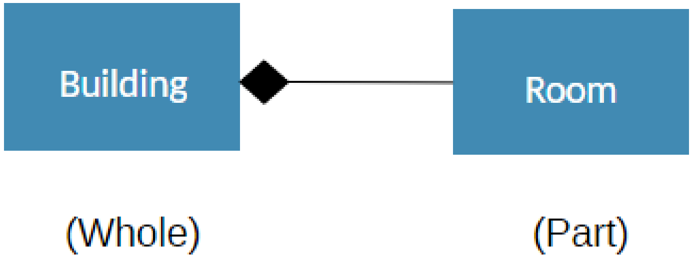
```
talk_show = AnimalTalkShow()
c = Cat()
talk_show.dias(c)
d = Dog()
talk_show.dias(d)
```

- These are special types of **has-a** relationship
- Aggregation: Part and whole relationship
  - Example: A team **has** players
    - Player **is a part** of team
    - However, if the team is dissolved, players may still exist
    - Team <aggregation> Player
- Composition: part is limited to the life time of whole
  - Building **has** rooms
    - Building is **made up of** room. If the building is destroyed, rooms are destroyed
    - Building <composition> Room

Aggregation



Composition



Dr Harsh Dev

e for Aggregation and Composition

```python
class Player:
    def __init__(self, name):
        self.name = name
class Team:
    def addPlayer(self,player):
        self.players.append(player)

    def __init__(self):
        self.players= []


p = Player("Sachin") #create player
t = Team() #create team
t.addPlayer(p) #add player to team
print(t.players[0].name)
del(t) #delete the team
print(p.name) #still the player exists
```

Player class with property: name

Team has Players: Team class with property: players[]

```python
class Room:
    pass

class Building:
    def __init__(self,room_count):
        self.rooms = []

        for i in range (0,room_count):
            r =  Room()
            self.rooms.append(r)

    def __del__(self):
        print("All rooms destroyed")
        del self.rooms
b = Building(3)
del(b)
```

A room class with no members

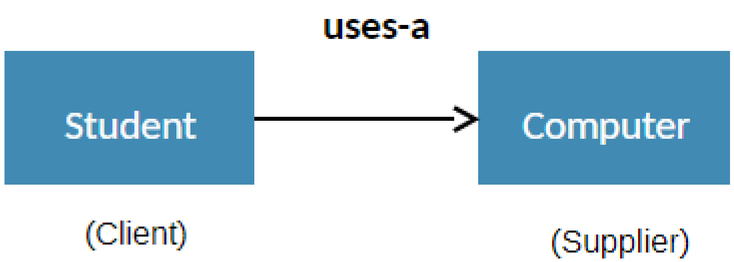
Building has rooms: Building class with property:room[]

__del__ method automatically called by the interpreter wher the instance is about to be destroyed

Delete the building object and all rooms inside gets destroyed as well

cenario:

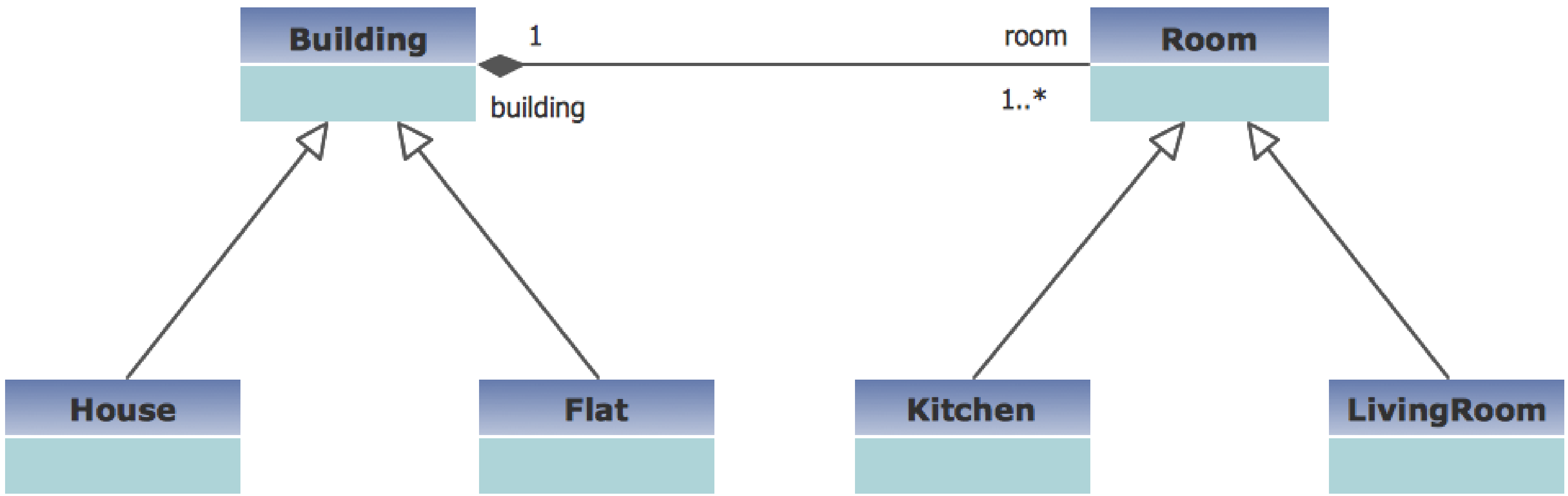
A student uses a Computer for lab assignments



Note that the Computer is not a part of Student. It is used only in *doLabAssignment* method for writing and executing assignment code.

```
class Computer:
    def writeCode(self,text):
        print(text,"written in editor")
    def execute(self):
        print("Code Executed")

class Student:
    def doLabAssignment(self,computer,assignmen
        computer.writeCode(assignment)
        computer.execute()

s = Student()
c = Computer()
s.doLabAssignment(c,"Assignment code")
```

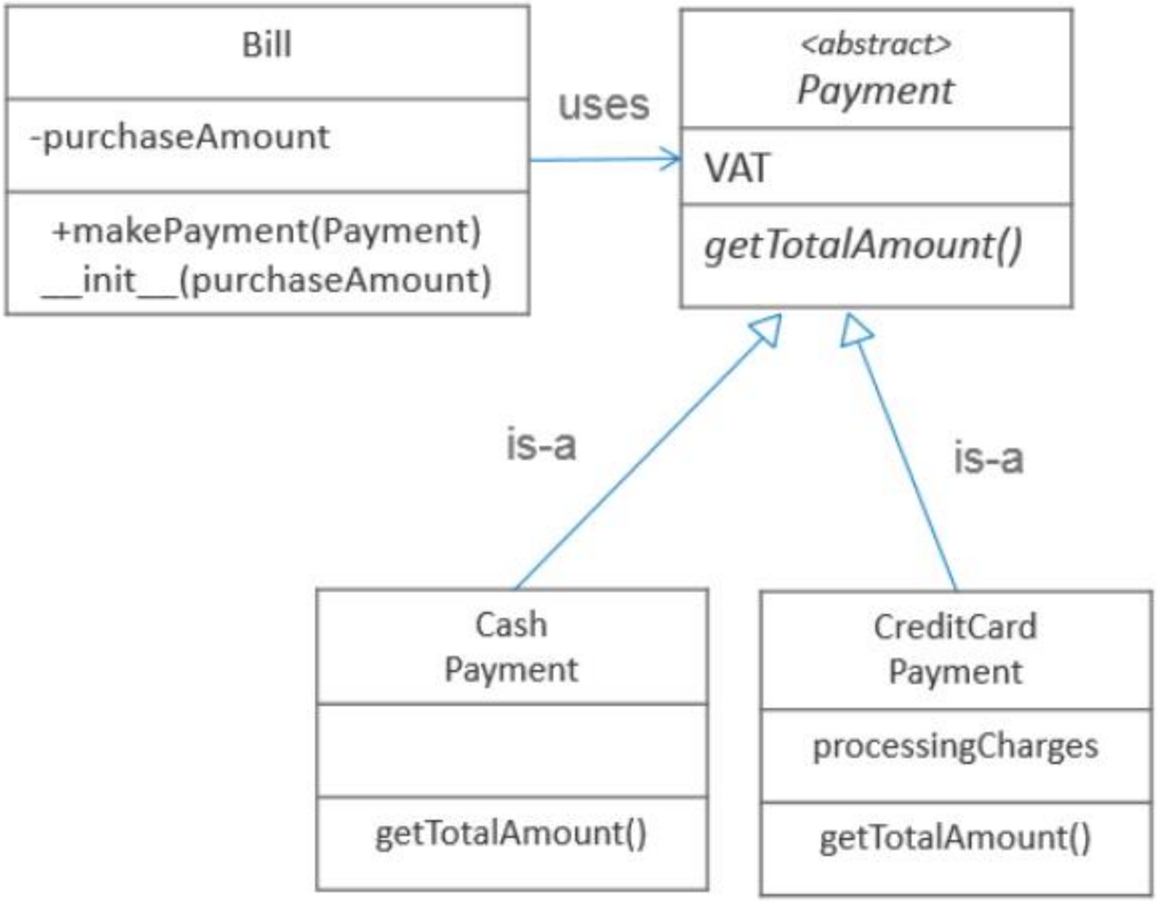
Student uses a methods of a Computer to write and execute a code


Building
1
building
room
1..*
Room
House
Flat
Kitchen
LivingRoom

Bill payment problem

- In a retail outlet there are **two modes of bill Payment**
- •**Cash :** Calculation includes VAT(15%)
- Total Amount = Purchase amount + VAT
- •**Credit card:** Calculation includes processing charge and VAT
- Total Amount = Purchase amount + VAT(15%)+
  Processing charge(2%)
- The act of bill payment is same but the formula used for calculation of total amount differs as per the mode of payment.

- **Q: Can the Payment maker simply call a method and that method dynamically selects the formula for the total amount?**
- **Demonstrate this Polymorphic behavior with code.**


Bill
-purchaseAmount
+makePayment(Payment)
__init__(purchaseAmount)
uses
<abstract>
Payment
VAT
getTotalAmount()
is-a
is-a
Cash
Payment
getTotalAmount()
CreditCard
Payment
processingCharges
getTotalAmount()

Continued..

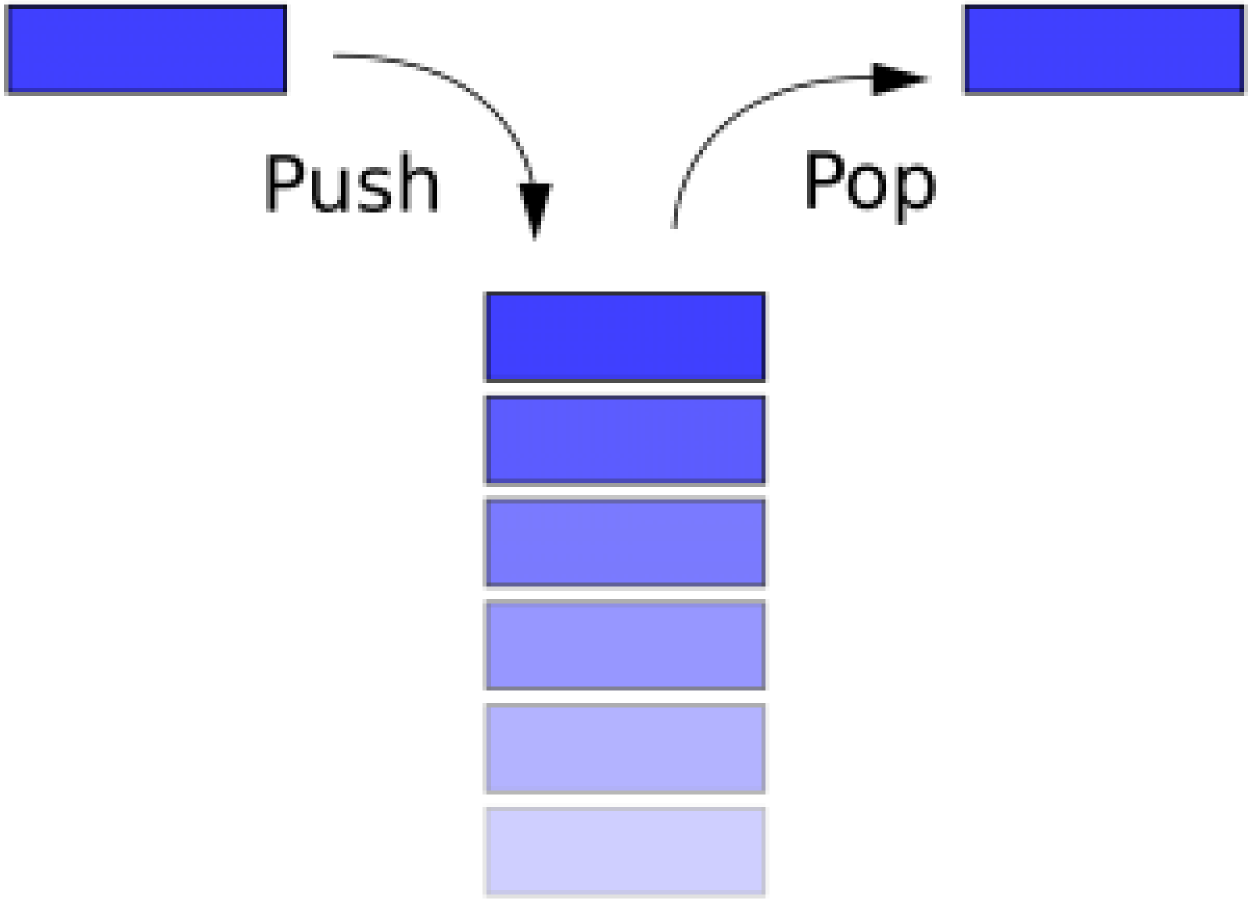
```python
from abc import ABC, abstractmethod

class Payment(ABC):

    VAT = 1.15

    @abstractmethod

    def totalAmount(self):

        pass

class CreditCardPayment(Payment):
    processingCharges = 1.02
    def getTotalAmount(self,purchaseAmt):
        amt = purchaseAmt * self.VAT #stmt1
        amt = amt * self.processingCharges
        return amt

class CashPayment(Payment):
    def getTotalAmount(self,purchaseAmt):
        return (purchaseAmt * self.VAT) #stmt2

class Bill:
    def __init__(self,purchaseAmount):
        self.__purchaseAmount = purchaseAmount
    def makePayment(self,mode):
        #Ensure that it is a valid mode of payment
        if (isinstance(mode, Payment)):
            #actual behavior is selected dynamically
            amount= mode.getTotalAmount(self.__purchaseAmount)
            print("Paid:", amount)
```

```python
#create a bill with
#purchaseAmount=1000
bill = Bill(1000)
cc = CreditCardPayment()
bill.makePayment(cc)
cash = CashPayment()
bill.makePayment(cash)
```

- Print(hasattr(c,'name'))

Data Structure

- A data structure is a specialized format for organizing and storing data.
- General data structure types include the array, the file, the record, the table, the tree, and so on.
- Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways.
- In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms.
- **Applications of Stack:**
- **1. Expression Evaluation**
- **2. Syntax Parsing**
- **3. Parenthesis Checking**
- **Function Call:** Stack is used to keep information about the active functions or subroutines.


Push
Pop

Abstract Classes

- Abstract classes are classes that contain one or more abstract methods.
- An abstract method is a method that is declared, but contains no implementation.
- Abstract classes may not be instantiated, and require subclasses to provide implementations for the abstract methods.
- Subclasses of an abstract class in Python are not required to implement abstract methods of the parent class.
- In fact, Python on its own doesn't provide abstract classes.
- Yet, Python comes with a module which provides the infrastructure for defining Abstract Base Classes (ABCs).
- This module is called - for obvious reasons - abc.

- A class that is derived from an abstract class cannot be instantiated unless all of its abstract methods are overridden.
- You may think that abstract methods can't be implemented in the abstract base class.
- This impression is wrong: An abstract method can have an implementation in the abstract class! Even if they are implemented, designers of subclasses will be forced to override the implementation.
- Like in other cases of "normal" inheritance, the abstract method can be invoked with super() call mechanism.
- This makes it possible to provide some basic functionality in the abstract method, which can be enriched by the subclass implementation.

```
from abc import ABC, abstractmethod
class AbstractClassExample(ABC):
        @abstractmethod
        def do_something(self):
                print("Some implementation!")
class AnotherSubclass(AbstractClassExample):
        def do_something(self):
                super().do_something()
                print("The enrichment from AnotherSubclass")

x = AnotherSubclass()
x.do_something()
```

Self and Reference Variable

mob2.return_product() can also be invoked as Mobile.return_product(mob2)
Thus self now refers to mob2, as this is actually pass by reference. For simplicity sake and for better readability we use mob2.return_product() instead of

```python
class Mobile:
    def __init__(self,price,brand):
        print (id(self))
        self.price = price
        self.brand = brand

    def return_product(self):
        print (id(self))
        print ("Brand being returned is ",self.brand," and price is ",self.price)

mob1 = Mobile(1000, "Apple")
print ("Mobile 1 has id", id(mob1))

mob2=Mobile(2000, "Samsung")
print ("Mobile 2 has id", id(mob2))

mob2.return_product()
Mobile.return_product(mob2)
```

OBJECT ORIENTED PROGRAMMING

Quiz 22

What is the output of the following code snippet?

```python
class Customer:
    def __init__(id,self,age):
        id.self=self
        id.age=age

c1=Customer(100,20)
print(c1.self)
```

Options:

- (•) 100
- ( ) 20
- ( ) Error

Submit

Your answer is Right

Here self is not the first parameter. Therefore the value 100 is assigned to self, which is the second a reference to the current object. Hence it creates an object with 2 attributes: self and age, where 20

Accessor and Mutator methods – Python

- ***A*** method defined within a class can either be an Accessor or a Mutator method.
- An Accessor method returns the information about the object, but do not change the state or the object.
- A Mutator method, also called an Update method, can change the state of the object.
- a = [1,2,3,4,5]
- a.count(1)
- a.index(2)
- a.append(6)
- The methods a.count() and a.index() are both Accessor methods since it doesn’t alter the object a in any sense, but only pulls the relevant information.
- But a.append() is a mutator method, since it effectively changes the object (list a) to a new one.