OBJECT-ORIENTED CONCEPT:

1. INHERITANCE(Sharing of Information)
2. ENCAPSULATION (Grouping of Information)
3. ABSTRACTION (Hiding of Information)
4. POLYMORPHISM (Redifining of Information)

INHERITANCE

Inheritance is a powerful feature in object oriented programming.

It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.

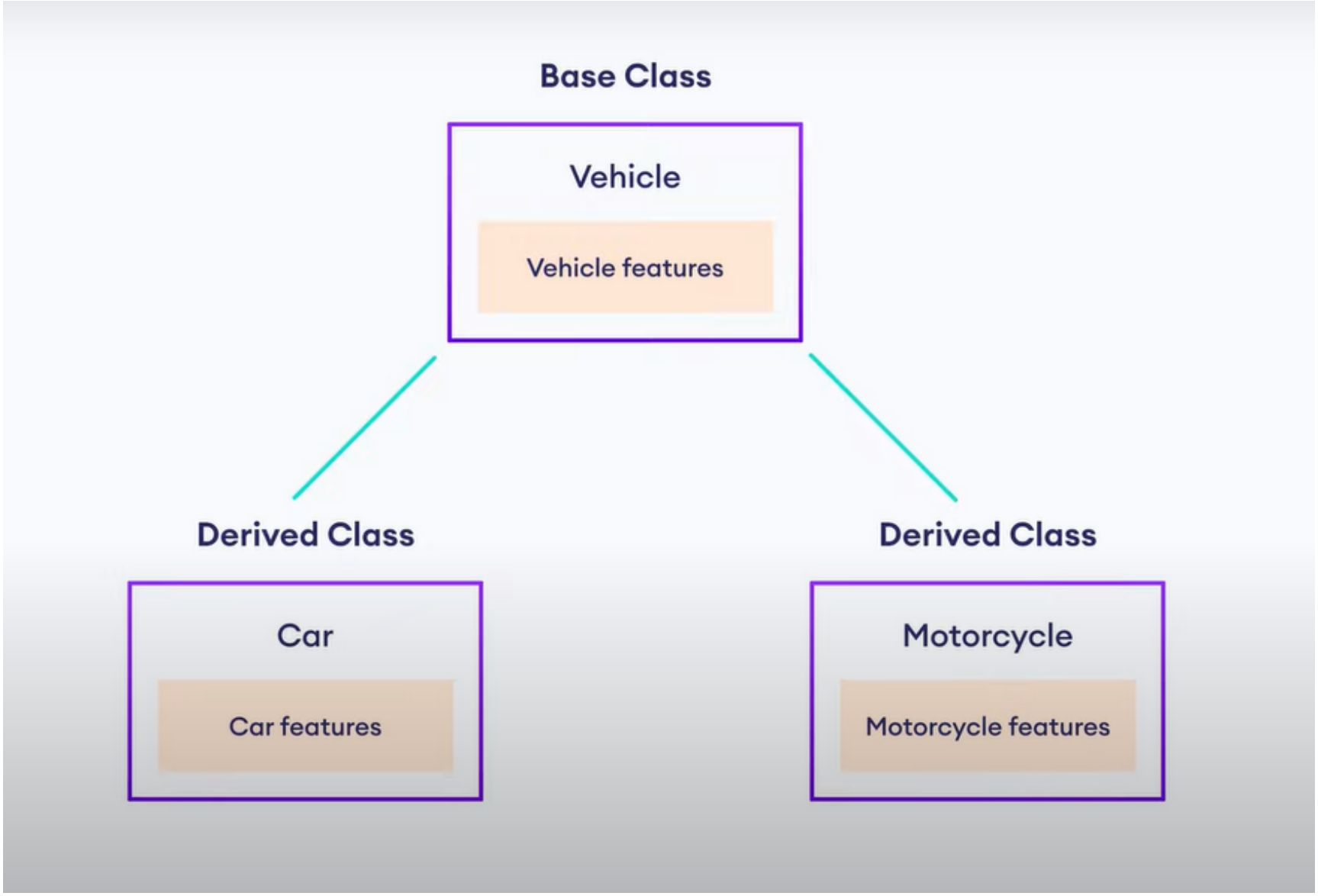
BENEFITS OF INHERITANCE:

1. Inheritance depicts relationships that resemble real-world scenarios.
2. It provides the feature of re-usability which allows the user to add more features to the derived class without altering it.
3. If a class Y inherits from class X, then automatically all the sub-classes of Y would inherit from class X.

Python Inheritance Syntax

```
class SuperClassName:
  Body of Super class


class DerivedClass_Name(SuperClass):
  Body of derived class
```

EXAMPLE:



In [ ]:
```
class Person:
    pass
print(Person.__bases__)
```

In [ ]:
```
class Person:
    pass
```

```
class Teacher(Person):
    pass

print(Teacher.__bases__)
```

In [ ]:

```
#SINGLE INHERITANCE
#METHODS ACCESSIBILITY

class A:
    def meth1(self):
        print("Inside Method 1")

    def meth2(self):
        print("Inside Method 2")

class B(A):
    def meth3(self):
        print("Inside Method 3")
    def meth4(self):
        print("Inside Method 4")

obj1=B()
obj1.meth1()
obj1.meth2()
obj1.meth3()


obj2=A()
obj2.meth3()
```

In [ ]:

```
#Attributes Accessibility

class A:
    def setdataA(self):
        self.attr1=90
        self.__attr2=80

    def check(self):
        print("Inside method 1")
        print("Attribute 1: ",self.attr1)
        print("Attribute 2: ",self.__attr2)
```

```
class B(A):
    def setdataB(self):
        self.attr3=70
        self.__attr4=60

    def check2(self):
        print("Inside method 2")
        print("Attribute 3: ",self.attr3)
        print("Attribute 4: ",self.__attr4)
        print("Attribute 1: ",self.attr1)

obj1=B()
obj1.setdataA()
obj1.check()

#obj1.setdataB()
obj1.check2()
```

In [ ]:

```
#Single Inheritance, Teacher class
#Inherting the person class

class Person:
    def setdetails(self,name,age):
        self.name=name
        self.age=age

    def Display(self):
        print("Name is:",self.name)
        print("Age is:",self.age)

class Teacher(Person):
    def setTeacherDetails(self,exp,r_area):
        self.exp=exp
        self.r_area=r_area


    def DisplayT(self):
        print("Experience is:",self.exp)
        print("Research Area is:",self.r_area)

Mohit=Teacher()
Mohit.setdetails("Mohit",'29')
Mohit.Display()
```

```
Mohit.setTeacherDetails('4','Graph Theory')
Mohit.DisplayT()
```

In [ ]:

```
#By default,the constructing method __init__() in the parent class is available to the child class.

class Father:
    def __init__(self):
        self.money=5000

        print("Father Class Constructor")

class Son(Father):
    def __init__(self):
        self.money=10000
        print("Son class Constructor")

    def display(self):
        print("Son Class Instance Method:",self.money)

S=Son()
S.display()
```

SUPER FUNCTION:

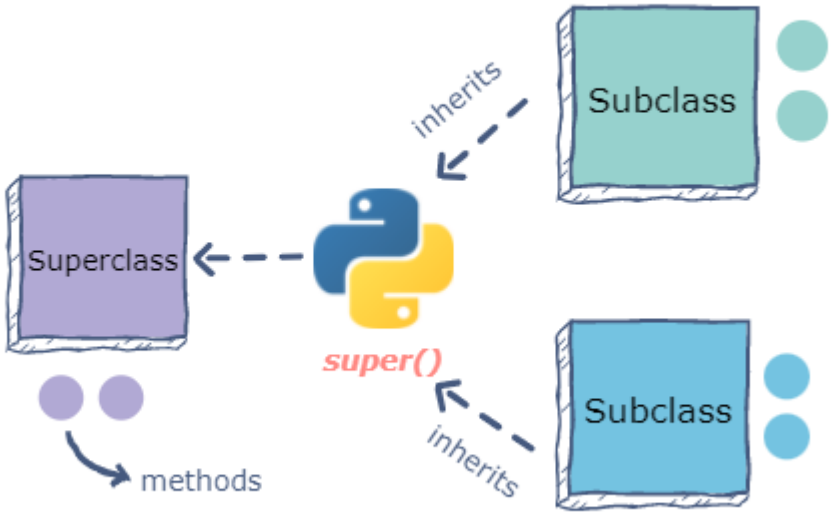
- The super() function in Python makes class inheritance more manageable and extensible. The function returns a temporary object that allows reference to a parent class by the keyword 'super'.

THE SUPER FUNCTION HAS TWO MAJOR USE:

1. To avoid the usage of the super (parent) class explicitly.
2. To enable multiple inheritance.

In [ ]:

```
help(super)
```



DEMONSTRATION OF SUPER FUNCTION

In [ ]:

```
#Program to demonstrate super function

class Rectangle:
    def __init__(self,length,width):
        self.length=length
        self.width=width

    def area(self):
        return self.length* self.width

    def perimeter(self):
        return self.length + 2 * self.width

class Square(Rectangle):
    def __init__(self,length):
        super().__init__(length,length)

obj1=Square(4)
print(obj1.area())
```

- Here, you've used super() to call the **init**() of the Rectangle class, allowing you to use it in the Square class without repeating code.
- Because the Square and Rectangle .**init**() methods are so similar, you can simply call the superclass's .**init**() method (Rectangle.**init**()) from that of Square by using super(). This sets the .length and .width attributes even though you just had to supply a single length parameter to the Square class.When you run this, even though your Square class doesn't explicitly implement it, the call to .area() will use the .area() method in the superclass and print 16. The Square class inherited .area() from the Rectangle class.

In [ ]:
```
#Constructor or methods with super() Method

class Father:
    def __init__(self):
        self.money=2000
        print("Father Class Constructor")

    def display(self):
        print("Father Class Instance method: ",self.money)


class Son(Father):
    def __init__(self):
        self.money=5000
        super().__init__()
        print("Son Class Constructor")

    def display(self):
        super().display()
        print("Son Class Instance Method: ",self.money)


S=Son()
S.display()
```

In [ ]:
```
#Single Inheritance using __init__() method

class Person:
    def __init__(self,name):
        self.name=name

    def printname(self):
```

```
        print("Name: ",self.name)

class Student(Person):
    def __init__(self,name,rollno):
        Person.__init__(self,name)
        self.rollno=rollno

    def display(self):
        Person.printname(self)
        print("Roll Number: ",self.rollno)

S1=Student("Ankit",8888)
S1.display()
```

In [ ]:

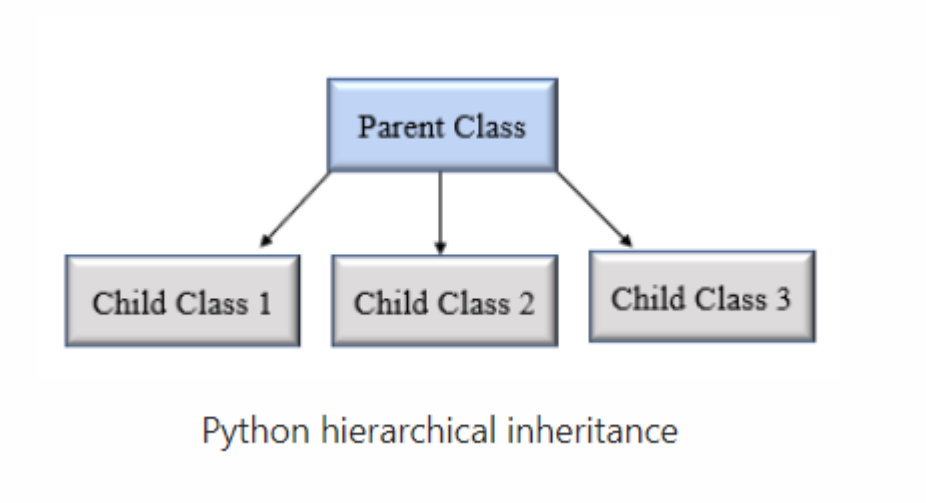
```
# Program to demonstrate Inheritance in class

class Animal:
    def legs(self):
        print("All have four legs")

class Dog(Animal):
    def sound(self):
        print("Bow Bow")

class Cat(Animal):
    def sound(self):
        print("Meow Meow")

labra=Dog()
labra.sound()
labra.legs()
```

TYPES OF INHERITANCE:

1. SINGLE INHERITANCE
2. HIERARCHICAL INHERITANCE
3. MULTI-LEVEL INHERITANCE
4. MULTIPLE INHERITANCE
5. HYBRID INHERITANCE

HIERARCHICAL INHERITANCE:

In Hierarchical inheritance, more than one child class is derived from a single parent class. In other words, we can say one parent class and multiple child classes.Both the child classes inherit properties from parent class and share the same property.



Python hierarchical inheritance

In [ ]:

```
# Python program to demonstrate
# Hierarchical inheritance


# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")

# Derived class1
class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")

# Derivied class2
class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")

# Driver's code
object1 = Child1()
```

```
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

In [ ]:

```
class Vehicle:
    def info(self):
        print("This is a vehicle.")

class Car(Vehicle):
    def car_info(self,name):
        print("Car: ",name)

class Bike(Vehicle):
    def bike_info(self,name):
        print("Bike: ",name)

c=Car()
c.car_info('KIA')

b=Bike()
b.bike_info('R15')
```

In [ ]:

```
class Vehicle:
    def info(self):
        print("This is a vehicle.")

class Car(Vehicle):
    def car_info(self,name):
        super().info()
        print("Car: ",name)

class Bike(Vehicle):
    def bike_info(self,name):
        print("Bike: ",name)

c=Car()
c.info()
c.car_info('KIA')
```

```
b=Bike()
b.bike_info('R15')
```

In [ ]:

```
#Hierarchial Inheritance:

class Person:

    def setDetails(self,name,age):
        self.name=name
        self.age=age

    def Displaydetails(self):
        print("Name: ",self.name)
        print("Age: ",self.age)


class Teacher(Person):

    def setTeacherDetails(self,exp,r_area):
        self.exp=exp
        self.r_area=r_area

    def DisplayTeacherDetails(self):
        print("Total Experience: ",self.exp)
        print("Research Area is: ",self.r_area)

class Student(Person):

    def setStudentDetails(self,course,branch):
        self.course=course
        self.branch=branch

    def DisplayStudentDetails(self):
        print("Course: ",self.course)
        print("Branch: ",self.branch)


T1=Teacher()
T1.setDetails('Mohit Kumar',29)
T1.Displaydetails()
T1.setTeacherDetails(4,'Graph Theory')
T1.DisplayTeacherDetails()

S1=Student()
```

```
S1.setDetails('Bad Boy Piyush',19)
S1.Displaydetails()
S1.setStudentDetails('B.Tech','Computer Science')
S1.DisplayStudentDetails()
```

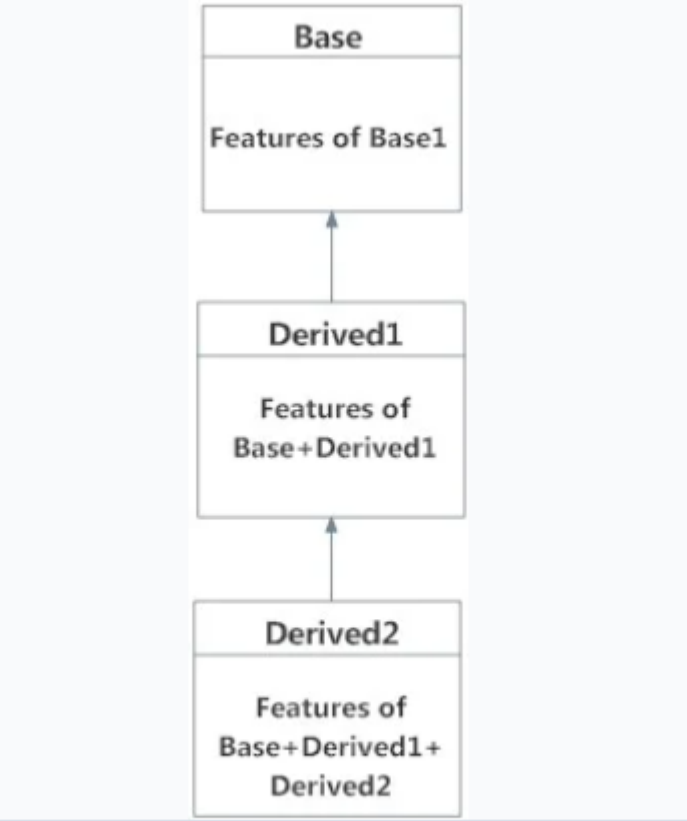
In [ ]:

```
#uSING sUPER() FUNCTION

class Person:

    def setDetails(self):
        self.name=input("Enter the name: ")
        self.age=input("Enter the age: ")

    def displaydetails(self):
        print("Name: ",self.name)
        print("Age: ",self.age)


class Teacher(Person):

    def setTeacherDetails(self):
        print("\n---Enter Teacher Details---")
        Person.setDetails(self)
        self.exp=float(input("Enter Experience in years: "))
        self.r_area=input("Enter Research Area: ")

    def displayTeacherDetails(self):
        print("\n----Teacher Details are as followed----")
        super().displaydetails()
        print("Experience :",self.exp)
        print("Research Area :",self.r_area)


class Student(Person):
    def setStudentDetails(self):
        print("\n---- Setting Student Details ----")
        Person.setDetails(self)
        self.course = input("Enter Course:")
        self.marks = []
        for i in range(1,4):
            self.marks.append(int(input(f"Enter marks of sub{i}::")))

    def displayStudentdetails(self):
        print("\n---- Getting Student Details ----")
```

```
        super().displaydetails()
        print("Course :",self.course)
        print("Marks::",dict(zip(["Maths","Python","Physics"],self.marks)))

T1 = Teacher()
T1.setTeacherDetails()
T1.displayTeacherDetails()
S1 = Student()
S1.setStudentDetails()
S1.displayStudentdetails()
```

3. MULTI-LEVEL INHERITANCE:

- The technique of deriving a class from already derived class is called "Multi-Level Inheritance".In multilevel inheritance, features of the base class and the derived class are inherited into the new derived class.



Multilevel Inheritance in Python

- SYNTAX:

  class Base:

  pass

  class Derived1(Base):

  pass

  class Derived2(Derived1):

  pass

In [ ]:
```
#Program to demonstrate multi-level inheritance
#Marks class inherits Student class & Result class inherits Marks class.

class Student:
    def setStudentDetails(self,rollno,name):
        self.rollno=rollno
        self.name=name

    def DisplayStudentDetails(self):
        print("Roll Number: ",self.rollno)
        print("Name: ",self.name)

class Marks(Student):

    def setMarks(self,m1,m2,m3):
        self.marks1=m1
        self.marks2=m2
        self.marks3=m3

    def displayMarks(self):
        print("Marks 1: ",self.marks1)
        print("Marks 2: ",self.marks2)
        print("Marks 3: ",self.marks3)

class Result(Marks):

    def calTotal(self):
        self.total=self.marks1+self.marks2+self.marks3

    def displayresult(self):
        print("Total Marks: ",self.total)

re=Result()
re.setStudentDetails(1,'kichlu')
re.setMarks(82,22,33)
re.DisplayStudentDetails()
re.displayMarks()
re.calTotal()
re.displayresult()
```

In [ ]:
```
# Python program to demonstrate
```

```
# multilevel inheritance

# Base class
class Grandfather:

    def __init__(self, grandfathername):
        self.grandfathername = grandfathername

# Intermediate class
class Father(Grandfather):
    def __init__(self, fathername, grandfathername):
        self.fathername = fathername

        # invoking constructor of Grandfather class
        Grandfather.__init__(self, grandfathername)

# Derived class
class Son(Father):
    def __init__(self,sonname, fathername, grandfathername):
        self.sonname = sonname

        # invoking constructor of Father class
        Father.__init__(self, fathername, grandfathername)

    def print_name(self):
        print('Grandfather name :', self.grandfathername)
        print("Father name :", self.fathername)
        print("Son name :", self.sonname)

# Driver code
s1 = Son('Swami','Mutthu Swami','Mutthu Swami Ayyer')
print(s1.grandfathername)
s1.print_name()
```

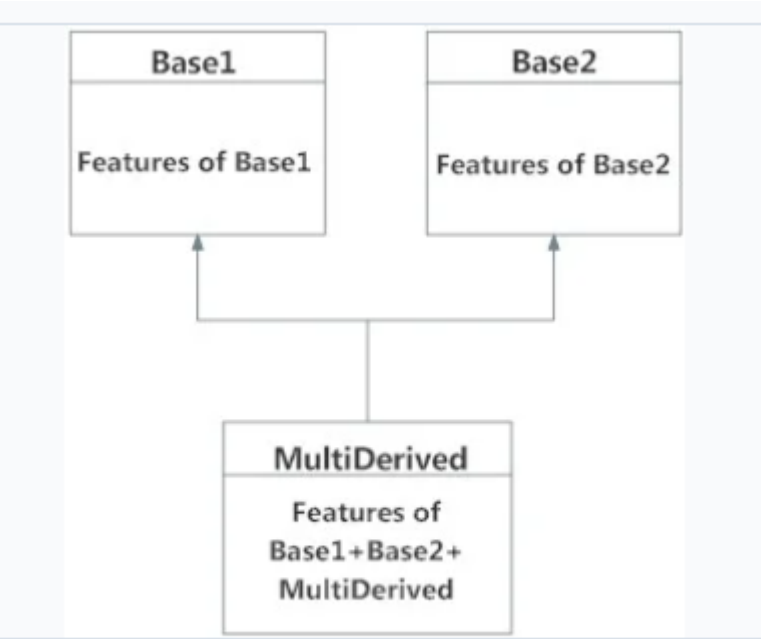
MULTIPLE INHERITANCE:

- When a derived class inherits features from more than one base class it is called "Multiple Inheritance". The derived class has all the features of both the base classes and in addition to them, can have additional new features.

SYNTAX:

class Base1: statement block class Base2: statement block class DErived (Base1,Base2): statement block

The figure below depicts the multiple inheritance



Multiple Inheritance in Python

- In the multiple inheritance scenario,any specified attribute is first searched in the current(or derived) class.
- If it is not found there, the search continues into parent classes using depth-first search technique, that is left-right fashion without searching same class twice.

NOTE: If the specified attribute is not found in the derived class, the search proceeds to look in the base class. This rule is applied recursively if the base class is derived from some other class.

In [ ]:

```
#Program to demonstrate multiple inheritance
#Result class inheriting both student and marks class

class Student:
    def setstudentdetails(self,rollno,name):
        self.rollno=rollno
        self.name=name
```

```
    def displaystudentdetails(self):
        print("Roll Number: ",self.rollno)
        print("Name: ",self.name)

class Marks:
    def setmarks(self,m1,m2,m3):
        self.marks1=m1
        self.marks2=m2
        self.marks3=m3

    def displaymarks(self):
        print("Marks 1: ",self.marks1)
        print("Marks 2: ",self.marks2)
        print("Marks 3: ",self.marks3)

class Result(Student,Marks):
    def caltotal(self):
        self.total=self.marks1+self.marks2+self.marks3

    def displayresult(self):
        print("Total MArks: ",self.total)

re=Result()
re.setstudentdetails(44,'kpmg')
re.setmarks(88,22,44)
re.displaystudentdetails()
re.displaymarks()
re.caltotal()
re.displayresult()
```

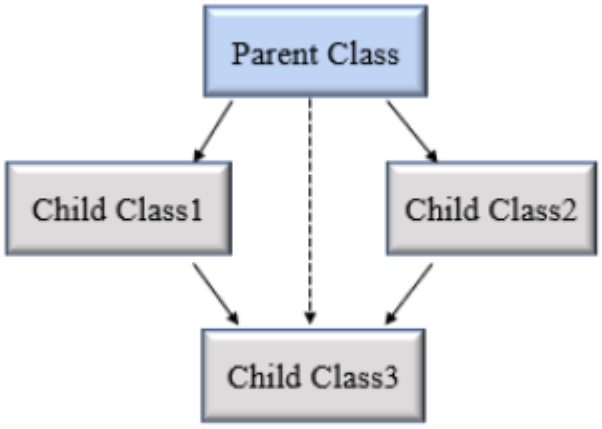
In [ ]:

```
# Python program to demonstrate
# multiple inheritance


# Base class1
class Mother:

    def mother(self):
        print(self.mothername)

# Base class2
class Father:

    def father(self):
```

```
        print(self.fathername)

# Derived class
class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)

# Driver's code
s1 = Son()
s1.fathername = "SAURAV"
s1.mothername = "REEMA"
s1.parents()
```

HYBRID INHERITANCE:

- An inheritance is said hybrid inheritance if more than one type of inheritance is implemented in the same code. This feature enables the user to utilize the feature of inheritance at its best. This satisfies the requirement of implementing a code that needs multiple inheritances in implementation.
- Mainly Hybrid inheritance is a combination of multiple inheritance and multilevel inheritance. The class is derived from the two classes as in the multiple inheritance. However, one of the parent classes is not the base class. It is a derived class!

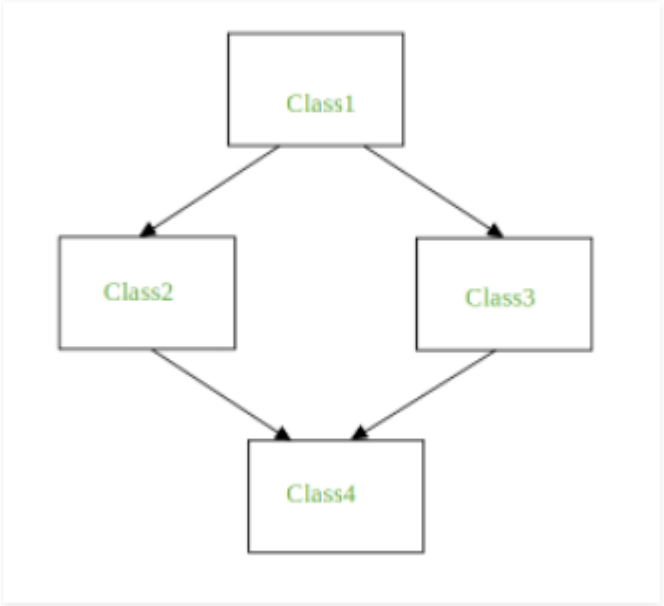


Python hybrid inheritance

In [ ]:

```
#Program to demonstrate hybrid inheritance
#Marks class inherits Student class
#Result class inherits Marks and Sports class

class Student:
    def setStudentDetails(self,rollno,name):
        self.rollno = rollno
        self.name = name

    def displayStudentDetails(self):
        print("Roll Number :",self.rollno)
        print("Name :",self.name)

class Marks(Student):
    def setMarks(self,m1,m2,m3):
        self.marks1 = m1
        self.marks2 = m2
        self.marks3 = m3

    def displayMarks(self):
        print("Marks 1 :",self.marks1)
        print("Marks 2 :",self.marks2)
        print("Marks 3 :",self.marks3)
```

```
class Sports:
    def setSportsMarks(self,pmarks):
        self.smarks = pmarks

    def displaySportsMarks(self):
        print("Sports Marks :",self.smarks)


class Result(Marks, Sports):
    def calTotal(self):
        self.total = self.marks1 + self.marks2 + self.marks3
        self.total = self.total + self.smarks


    def displayResult(self):
        print("Total marks :",self.total)


re = Result()
re.setStudentDetails(1,"Aman")
re.setMarks(88,98,92)
re.setSportsMarks(95)
re.displayStudentDetails()
re.displayMarks()
re.displaySportsMarks()
re.calTotal()
re.displayResult()
```



DIAMOND PROBLEM:

It refers to an ambiguity that arises when two classes Class2 and Class3 inherit from a superclass Class1 and class Class4 inherits from both Class2 and Class3. If there is a method “m” which is an overridden method in one of Class2 and Class3 or both then the ambiguity arises which of the method “m” Class4 should inherit.

- Diamond problem is a problem due to multiple or hybrid inheritance.
- It is a relationship exist when at least one of the parent classes can be accessed through multiple path from the bottommost class.
- The bottommost class inherits the members of the base class via different path so that it creates ambiguity or create duplicate members.
- This ambiguity must be avoided. To prevent base classes from being accessed more than once, the dynamic algorithm linearizes the search order in such a way that the left-to right ordering specified in each class is preserved.

In [ ]:
```
class A:
    def learn(self):
        print("This is method 1 in class A")

class B:
    def learn(self):
        print("This is method 2 in class B")

class C:
```

```
    def learn(self):
        print("This is method 3 in class C")

class D(C,B):
    pass

a=A()
b=B()
c=C()
d=D()

d.learn()
a.learn()
```

MRO: METHOD RESOLUTION ORDER

- In Python, every class whether built-in or user-defined is derived from the object class and all the objects are instances of the class object. Hence, the object class is the base class for all the other classes.
- In the case of multiple inheritance, a given attribute is first searched in the current class if it's not found then it's searched in the parent classes. The parent classes are searched in a left-right fashion and each class is searched once.
- If we see the above example then the order of search for the attributes will be Derived, Base1, Base2, object. The order that is followed is known as a linearization of the class Derived and this order is found out using a set of rules called "Method Resolution Order (MRO)."

In [3]:

```
#MRO

class A:
    def ping(self):
        print("Ping: Inside class A")

class B(A):
    def pong(self):
        print("Pong: Inside class B")

class C(A):
    def ping(self):
        print("Ping: Inside class C")

class D(B,C):
```

```
    def ping(self):
        super().ping()
        print("Post-Ping: Inside class D")

    def pingpong(self):
        self.ping()
        super.ping()
        self.pong()
        super().pong()
        C.pong(self)

d=D()
d.pong()

#Printing MRO
print("\nMethod Resolution Order is: \n")
for i in D.mro():
    print(i)
```

```
Pong: Inside class B

Method Resolution Order is:

<class '__main__.D'>
<class '__main__.B'>
<class '__main__.C'>
<class '__main__.A'>
<class 'object'>
```

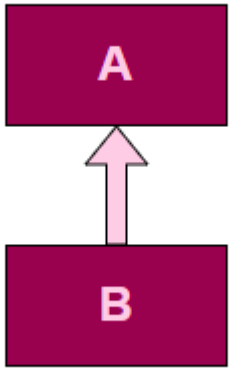
MRO is a concept used in inheritance. It is the order in which a method is searched for in a classes hierarchy and is especially useful in Python because Python supports multiple inheritance.

In Python, the MRO is from bottom to top and left to right. This means that, first, the method is searched in the class of the object. If it's not found, it is searched in the immediate super class. In the case of multiple super classes, it is searched left to right, in the order by which was declared by the developer.

EXAMPLE 1: class B(A)

- In this case, the MRO would be B -> A.

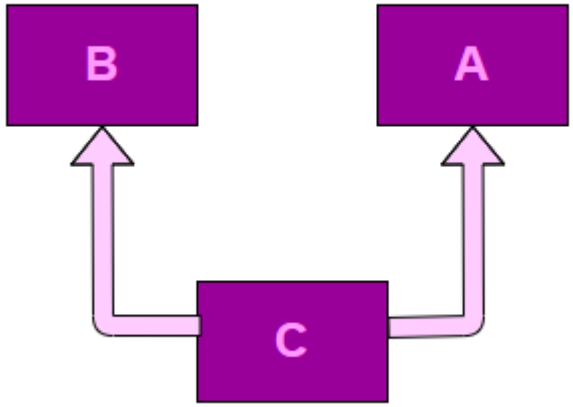
- Since B was mentioned first in the class declaration, it will be searched for first while resolving a method.



In [4]:
```
class A:
  def method(self):
    print("A.method() called")

class B(A):
  def method(self):
    print("B.method() called")

b = B()
b.method()
```

```
B.method() called
```

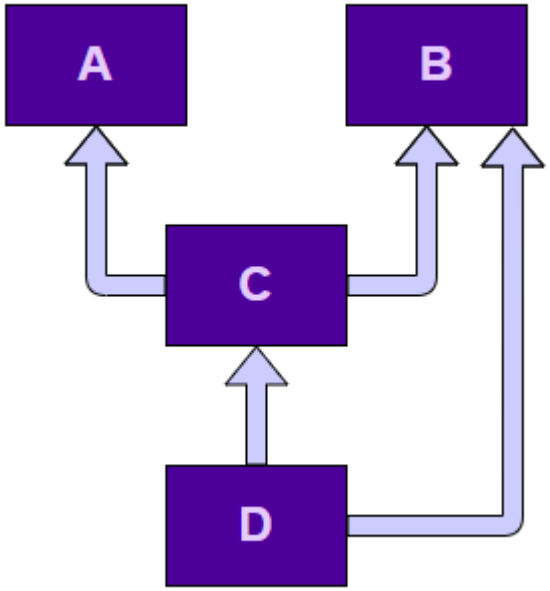
This is a simple case with single inheritance. In this case, when b.method() is called, it first searches for the method in class B. In this case, class B had defined the method; hence, it is the one that was executed. In the case where it is not present in B, then the method from its immediate super class (A) would be called. So, the MRO for this case is: B -> A



EXAMPLE 2:

In [ ]:

```
class A:
  def method(self):
    print("A.method() called")

class B:
  pass

class C(B, A):
  pass

c = C()
c.method()


#The MRO for this case is:
#C -> B -> A
#The method only existed in A, where it was searched for last.
```



EXAMPLE 3:

In [ ]:

```
class A:
  def method(self):
    print("A.method() called")

class B:
  def method(self):
    print("B.method() called")

class C(A, B):
  pass

class D(C, B):
  pass

d = D()
d.method()
```

The MRO for this can be a bit tricky. The immediate superclass for D is C, so if the method is not found in D, it is searched for in C. However, if it is not found in C, then you have to decide if you should check A (declared first in the list of C's super classes) or check B (declared in D's list of super classes after C). In Python 3 onwards, this is resolved as first checking A. So, the MRO becomes:

D -> C -> A -> B

ABSTRACT CLASS:

- In object-oriented programming, an abstract class is a class that cannot be instantiated. However, you can create classes that inherit from an abstract class.
- We use an abstract class to create a blueprint for other classes.
- Similarly, an abstract method is an method without an implementation. An abstract class may or may not include abstract methods.
- Python doesn't directly support abstract classes. But it does offer a module that allows you to define abstract classes.

WHY WE USE ABSTRACT CLASS?

By defining an abstract base class, you can define a common Application Program Interface(API) for a set of subclasses. This capability is especially useful in situations where a third-party is going to provide implementations, such as with plugins, but can also help you when working in a large team or with a large code-base where keeping all classes in your mind is difficult or not possible.

In [12]:
```
from abc import ABC,abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        return 0


class Rectangle(Shape):

    def __init__(self):
        self.length=10
        self.breadth=12

    def area(self):
        return self.length*self.breadth

rec=Rectangle()
print(rec.area())
```

```
120
```

In [ ]:

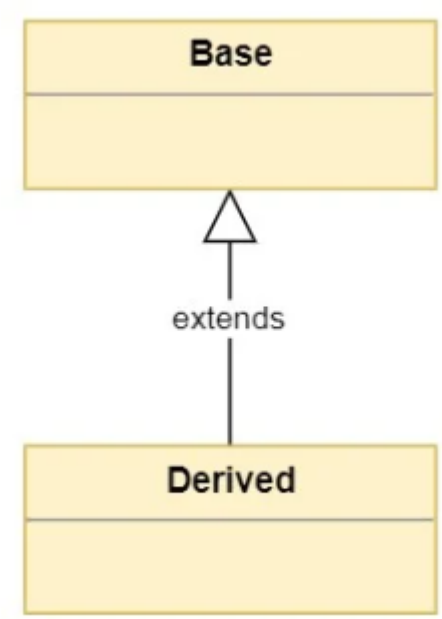
```
#Program to demonstrate abstract class

from abc import ABC, abstractmethod

class Polygon(ABC):

    @abstractmethod
    def noofsides(self):
        pass

class Triangle(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 3 sides")

class Pentagon(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 5 sides")

class Hexagon(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 6 sides")

class Quadrilateral(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 4 sides")

# Driver code
R = Triangle()
R.noofsides()

K = Quadrilateral()
K.noofsides()
```

```
R = Pentagon()
R.noofsides()

K = Hexagon()
K.noofsides()
```

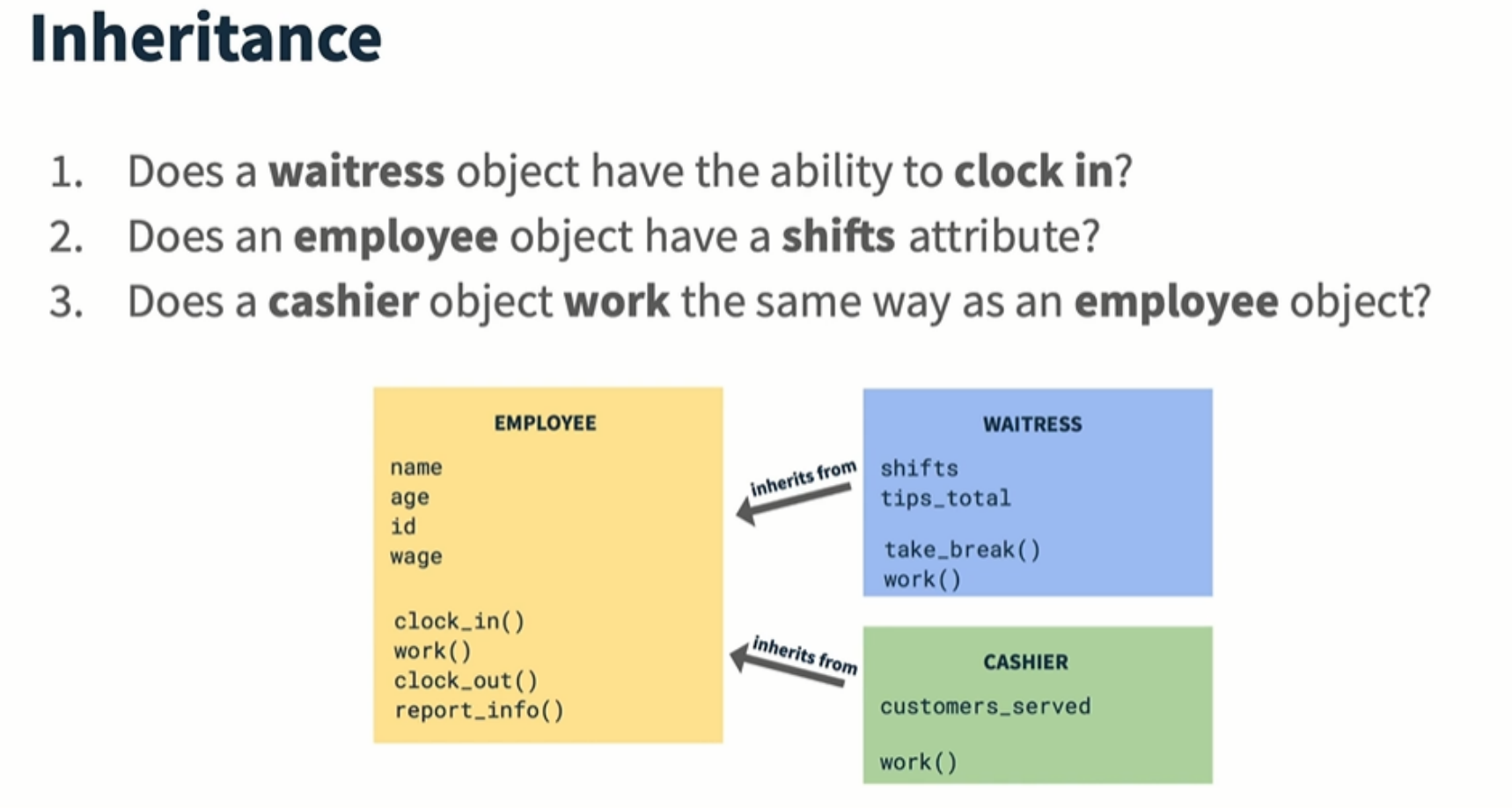
Inheritance and composition are two important concepts in object oriented programming that model the relationship between two classes. They are the building blocks of object oriented design, and they help programmers to write reusable code.

WHAT IS INHERITANCE?

Inheritance models what is called an **is a** relationship. This means that when you have a Derived class that inherits from a Base class, you created a relationship where Derived is a specialized version of Base.

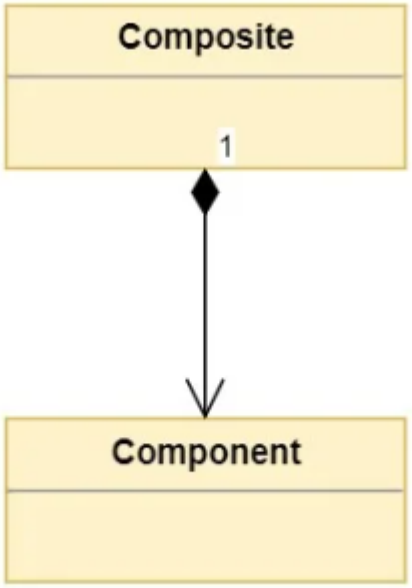


Classes are represented as boxes with the class name on top. The inheritance relationship is represented by an arrow from the derived class pointing to the base class. The word **extends** is usually added to the arrow.



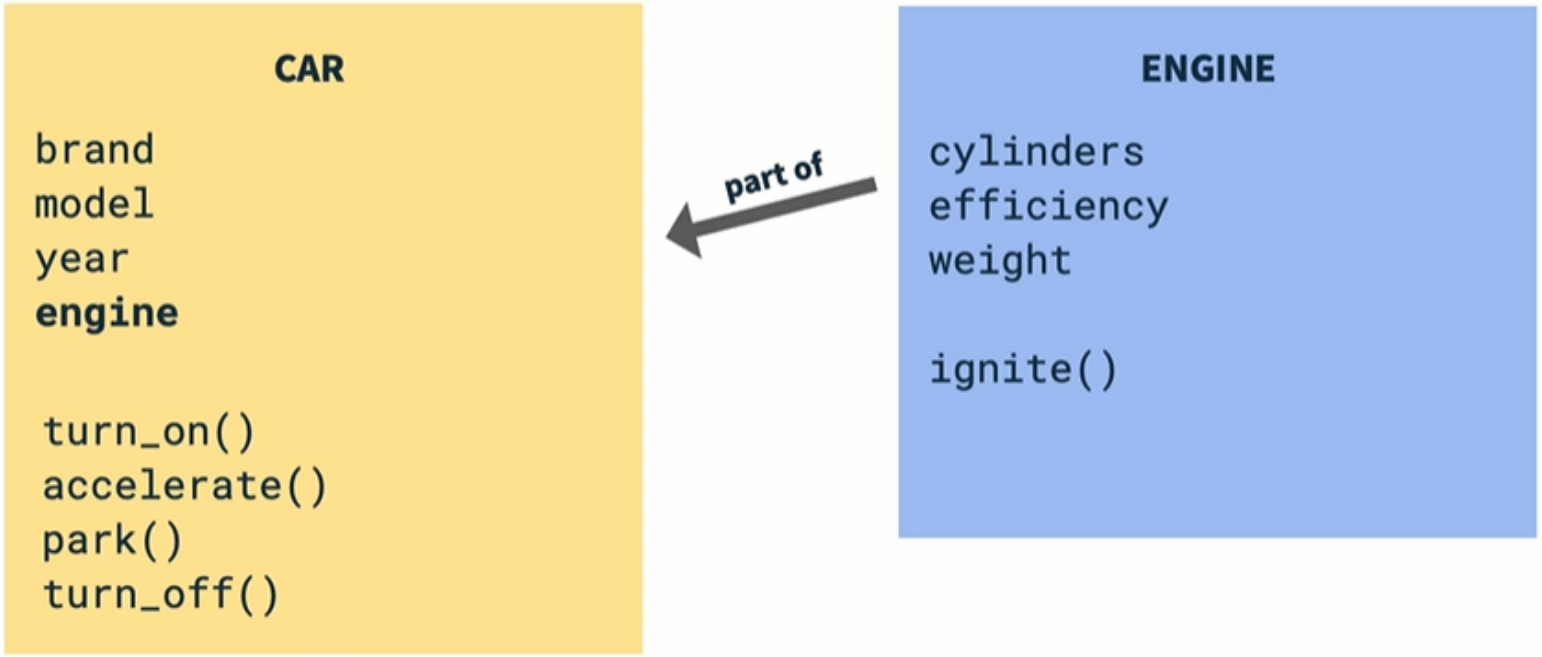
WHAT IS COMPOSTION/CONTAINERSHIP/COMPLEX OBJECT?

Composition is a concept that models a **has a** relationship. It enables creating complex types by combining objects of other types. This means that a class Composite can contain an object of another class Component. This relationship means that a Composite **has a** Component.



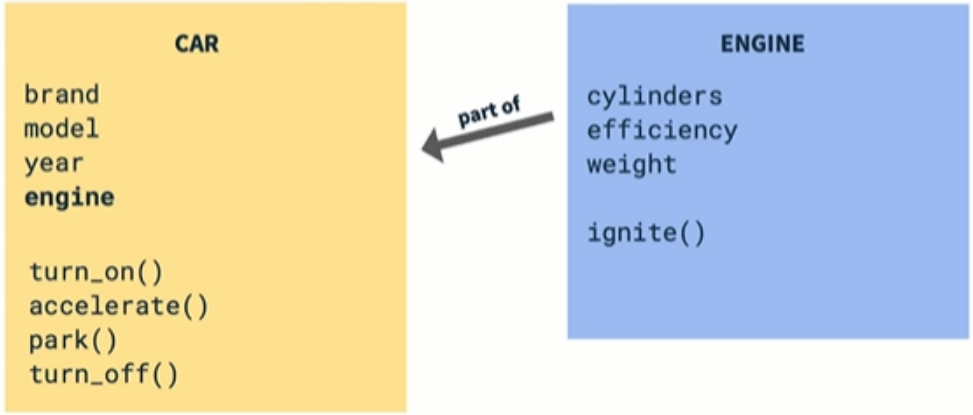
Composition is represented through a line with a diamond at the composite class pointing to the component class. The composite side can express the cardinality of the relationship. The cardinality indicates the number or valid range of Component instances the Composite class will contain.

Composition



Composition

1. What is the type of the **engine** attribute?
2. Does the **accelerate()** method have access to the **efficiency** attribute?
3. Can the **ignite()** method in the **engine** class access the **brand** attribute?



In [3]:

```
#Program to demonstrate Containership

class Marks:
    def __init__(self,m1,m2,m3):
        self.marks1 = m1
        self.marks2 = m2
        self.marks3 = m3
    def showmarks(self):
        print('Marks 1 :',self.marks1)
        print('Marks 2 :',self.marks2)
        print('Marks 3 :',self.marks3)
class Student:
    def __init__(self,roll,name,m1,m2,m3):
        self.roll = roll
        self.name = name
        self.marks = Marks(m1,m2,m3)
```

```
    def display(self):
        print('Roll Number :',self.roll)
        print('Name :',self.name)
        self.marks.showmarks()
S = Student(101,'ABC',89,91,78)
S.display()
```

```
Roll Number : 101
Name : ABC
Marks 1 : 89
Marks 2 : 91
Marks 3 : 78
```

In [8]:

```
#Containership

class Salary:
    def __init__(self, pay):
        self.pay = pay

    def get_total(self):
        return (self.pay*12)


class Employee:
    def __init__(self, pay, bonus):  # pay object is contained in another class Employee
        self.pay = pay
        self.bonus = bonus
        self.obj_salary = Salary(self.pay)

    def annual_salary(self):
        return "Total: " + str(self.obj_salary.get_total() + self.bonus)  #Execution of containership


obj_emp = Employee(600, 500)
print(obj_emp.annual_salary())
```

```
Total: 7700
```

In [9]:

```
#Composition/Containership

class Component:

# composite class constructor
    def __init__(self):
```

```
        print('Component class object created...')

    # composite class instance method
    def m1(self):
        print('Component class m1() method executed...')


class Composite:

    # composite class constructor
    def __init__(self):

        # creating object of component class
        self.obj1 = Component()

        print('Composite class object also created...')

    # composite class instance method
    def m2(self):

        print('Composite class m2() method executed...')

        # calling m1() method of component class
        self.obj1.m1()


# creating object of composite class
obj2 = Composite()

# calling m2() method of composite class
obj2.m2()
```

```
Component class object created...
Composite class object also created...
Composite class m2() method executed...
Component class m1() method executed...
```

Comparison between Inheritance and Containership

INHERITANCE:

1. Enables a class to inherits data and functions from a base class by extending it.

2. The derived class may override the funcionality of base class.
3. The derived class may add data or functions to the base class.
4. Inheritance represents **IS-A** relationship.
5. **Example: Student is a person**

CONTAINERSHIP:

1. Enables a class to contain object of different classes as its data member.
2. The container class cannot alter or override the functionality of the contained class.
3. The container class cannot add anything to the contained class.
4. Containership represents **HAS-A** relationship.
5. **Example: class One has a class Two**

POLYMORPHISM IN PYTHON:

1. ***Polymorphism, one of the essential concepts of OOP, it refers to having several different forms.***
2. *Inheritance is related to classes and their hierarchy, polymorphism on the other hand is related to methods.*
3. ***Polymorphism is a concept that enables the programmers to assign a different meaning or usage to a method in different context.***

POLYMORPHISM exists when a number of subclasses is defined which have methods of same name. A function can use objects of any of the polymorphic classes irrespective of the fact that these classes are individually distinct.