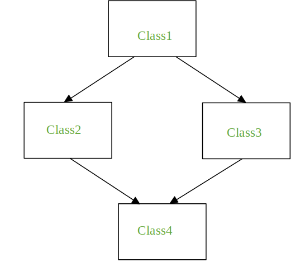
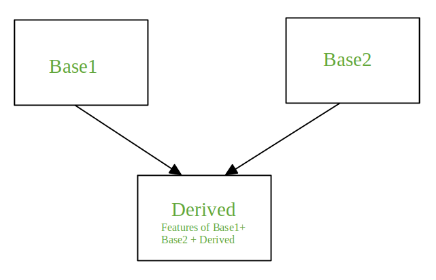
**Q1. What is the meaning of multiple inheritance?**

When a class is derived from more than one base class it is called multiple Inheritance. The derived class inherits all the features of the base case.



the problem faced during multiple inheritance is 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.

**Q2. What is the concept of delegation?**

Delegation is an object oriented technique (also called a design pattern). Let's say you have an object x and want to change the behaviour of just one of its methods. You can create a new class that provides a new implementation of the method you're interested in changing and delegates all other methods to the corresponding method of x.

Python programmers can easily implement delegation. For example, the following class implements a class that behaves like a file but converts all written data to uppercase:

class UpperOut:

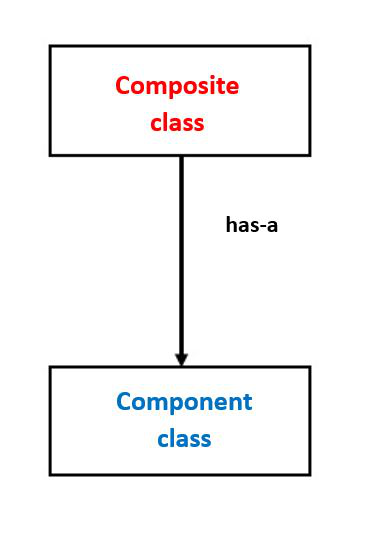
    def \_\_init\_\_(self, outfile):  
        self.\_outfile = outfile

    def write(self, s):  
        self.\_outfile.write(s.upper())

    def \_\_getattr\_\_(self, name):  
        return getattr(self.\_outfile, name)

**Q3. What is the concept of composition?**

**Composition** is an object oriented design concept that models a **has a** relationship. In composition, a class known as **composite** contains an object of another class known to as **component**. In other words, a composite class **has a** component of another class.



Composition allows composite classes to reuse the implementation of the components it contains. The composite class doesn’t inherit the component class interface, but it can leverage its implementation.

The composition relation between two classes is considered loosely coupled. That means that changes to the component class rarely affect the composite class, and changes to the composite class never affect the component class.

This provides better adaptability to change and allows applications to introduce new requirements without affecting existing code.

**Q4. What are bound methods and how do we use them?**

If a function is an attribute of class and it is accessed via the instances, they are called bound methods. A bound method is one that has ‘[self](https://www.geeksforgeeks.org/self-in-python-class/)‘ as its first argument. Since these are dependent on the instance of classes, these are also known as instance methods.

**class** sample(object):

    # Static variable for object number

    objectNo **=** 0

**def** \_\_init\_\_(self, name1):

        # variable to hold name

        self.name **=** name1

        # Increment static variable for each object

        sample.objectNo **=** sample.objectNo **+** 1

        # each object's unique number that can be

        # considered as ID

        self.objNumber **=** sample.objectNo

**def** myFunc(self):

**print**("My name is ", self.name,

              "from object ", self.objNumber)

**def** alterIt(self, newName):

        self.name **=** newName

**def** myFunc2():

        print("I am not a bound method !!!")

# creating first instance of class sample

samp1 **=** sample("A")

samp1.myFunc()

# creating second instance of class sample

samp2 **=** sample("B")

samp2.myFunc()

samp2.alterIt("C")

samp2.myFunc()

samp1.myFunc()

**Output:**

My name is A from object 1

My name is B from object 2

My name is C from object 2

My name is A from object 1

**Q5. What is the purpose of pseudoprivate attributes?**

 Pseudoprivate attributes are also useful in larger frameworks or tools, both to avoid introducing new method names that might accidentally hide definitions elsewhere in the class tree and to reduce the chance of internal methods being replaced by names defined lower in the tree. If a method is intended for use only within a class that may be mixed into other classes, the double underscore prefix ensures that the method won't interfere with other names in the tree, especially in multiple-inheritance scenarios

Pseudoprivate names also prevent subclasses from accidentally redefining the internal method's names,