Q1. Explain Class and Object with respect to Object-Oriented Programming. Give a suitable example.

Python Classes

A class is considered as a blueprint of objects. We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

In [5]:
```
# create class

class Student:
    name = ""
```

Since many houses can be made from the same description, we can create many objects from a class.

Python Objects

An object is called an instance of a class. For example, suppose Student is a class then we can create objects like Student1, Student2, etc from the class.

In [9]:
```
# create objects of class

Student1 = Student().name="Virat"

Student1
```

Out[9]:
```
'Virat'
```

Q2. Name the four pillars of OOPs.

The four pillars of OOP or Object Oriented Programming are as follows:

1. Inheritance.
2. Encapsulation.
3. Polymorphism.
4. Abstraction.

1. Inheritance in python

In case of real world objects, every element is a specialized within its general group of elements.

For example, if we have a general class "Animal", it can have two specialized classes like "Wild" and "Domestic". Under "Domestic" also we may have multiple other specialized class. Thus "Wild" is a specialized subclass of the parent "Animal" class and the properties

of "Wild" class share some properties with its parent class but it also has some unique properties which makes it unique within in parents class. This is what is called "Inheritance".

Lets see example of inheritance in python:

In [10]:
```
# Creating a Class in Python
class Animal:
   def sounds(self):
     print("This animals makes some sound!")

# Wild class inherits Animal class
class Wild(Animal):
  def asPet(self):
    print("This animal can't be pet!")

# Domestic class inherits Animal class
class Domestic(Animal):
  def asPet(self):
    print("This animal can be pet!")

# Creating object of Wild class
tiger = Wild()

tiger.sounds()
# Output : This animals makes some sound!

tiger.asPet()
# Output : This animal can't be pet!
```

```
This animals makes some sound!
This animal can't be pet!
```

2. Encapsulation in python

In Python, we can protect the member variables of a class from being accessed by any program outside our class using the priniciple of Encapsulation.

Encapsulation means binding up of code and data together under 1 wrapper which we already know as "Class" and provide proper access to change or modify the data of the class. We hide the variables by providing one underscore as prefix to the name of the variable.

Let see an example of encapsulation in python:

In [11]:
```
# Creating a Class in Python
class Animal:

  # Class Constructor
  def __init__(self,species):
     self._species = species

  def sounds(self):
     print("This animals makes some sound!")


tiger = Animal("Tiger")
print(tiger._species)
```

```
# Output : Tiger
```

```
Tiger
```

3. Polymorphism in python

The word "Polymorphism" has been derived from Greek literature meaning "having many forms". It generally emphasized on the fact of common interface for general set of activities.

An example of polymorphism in python is given below.

In [12]:
```
# Creating a Class in Python
class Dog:

  def makeSound(self):
    print("Dogs bark!")

class Tiger:

  def makeSound(self):
    print("Tigers growl!")

# common interface
def check_makeSound(animal):
    animal.makeSound()

myDog = Dog()
myTiger = Tiger()

check_makeSound(myDog) # Output : Dogs bark!
check_makeSound(myTiger) # Output : Tigers growl!
```

```
Dogs bark!
Tigers growl!
```

4. Abstraction in python

The concept of abstraction in O.O.P. is mainly utilized to create a blueprint of a class.

Like we just define what member methods should be present in the class and not the way how the class implements them. The abstract class just defines what methods should be present in the classes which inherits it and not what the method does. The activity of the method is left on the subclass to decide which inherits the abstract class.

By default, Python doesn't provide abstraction. There is a module named "abc" (Abstract Base Class) which helps us to realize the concept of abstraction in Python.

In [13]:
```
# Abstraction
from abc import ABC, abstractmethod

class Book(ABC):

  # Abstract method
  def book_name(self):
    pass

class Python(Book):
```

```
  # Overwrite Abstract method
  def book_name(self):
    print("This is a Python programming book!")

class Java(Book):

  # Overwrite Abstract method
  def book_name(self):
    print("This is a Java programming book!")

myBook = Java()
myBook.book_name()

# Output : This is a Java programming book!
```

```
This is a Java programming book!
```

Q3. Explain why the __init__() function is used. Give a suitable example.

All classes have a function called __init__(), which is always executed when the class is being initiated.

__init__() function is used to assign values to object properties, or other operations that are necessary to do when the object is being created:

In [14]:
```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

```
John
36
```

Q4. Why self is used in OOPs?

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class

In [16]:
```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)
```

```
p1 = Person("John", 36)
p1.myfunc()
```

```
Hello my name is John
```

Q5. What is inheritance? Give an example for each type of inheritance.

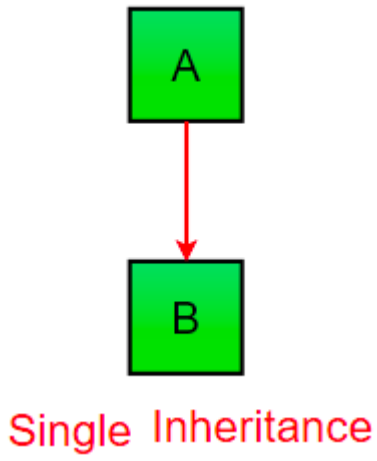
Inheritance is defined as the mechanism of inheriting the properties of the base class to the child class.

Types of Inheritance in Python

Types of Inheritance depend upon the number of child and parent classes involved. There are five types of inheritance in Python:

1) Single Inheritance:

Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.

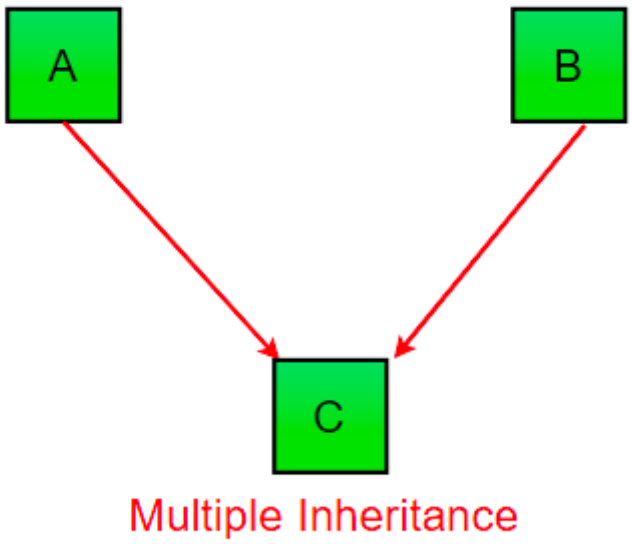


In [17]:
```
# Python program to demonstrate
# single inheritance

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

# Derived class


class Child(Parent):
    def func2(self):
        print("This function is in child class.")


# Driver's code
object = Child()
object.func1()
object.func2()
```

```
This function is in parent class.
This function is in child class.
```

2) Multiple Inheritance:

When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class.



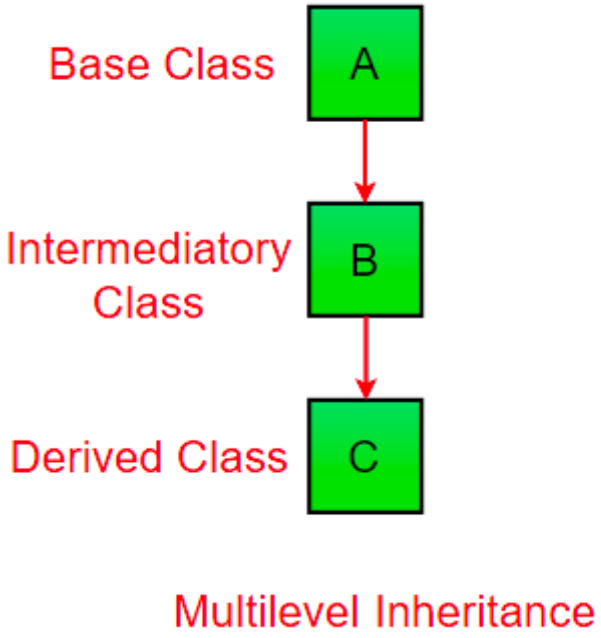
In [19]:
```python
# Python program to demonstrate
# multiple inheritance

# Base class1
class Mother:
    mothername = ""

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

# Base class2


class Father:
    fathername = ""

    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 = "Rahul"
```

```
s1.mothername = "Sangita"
s1.parents()
```

```
Father : Rahul
Mother : Sangita
```

3) Multilevel Inheritance :

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and a grandfather.



Multilevel Inheritance

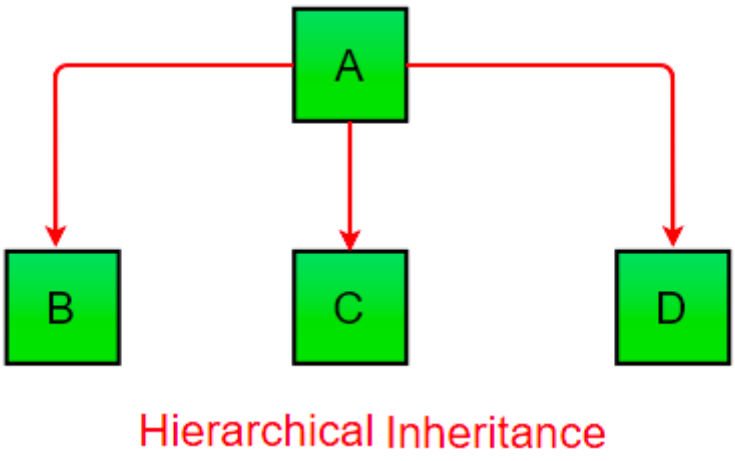
In [20]:
```
# Python program to demonstrate
# multilevel inheritance

# 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('Prince', 'Rampal', 'Lal mani')
print(s1.grandfathername)
s1.print_name()
```

```
Lal mani
Grandfather name : Lal mani
Father name : Rampal
Son name : Prince
```

4) Hierarchical Inheritance:

When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.



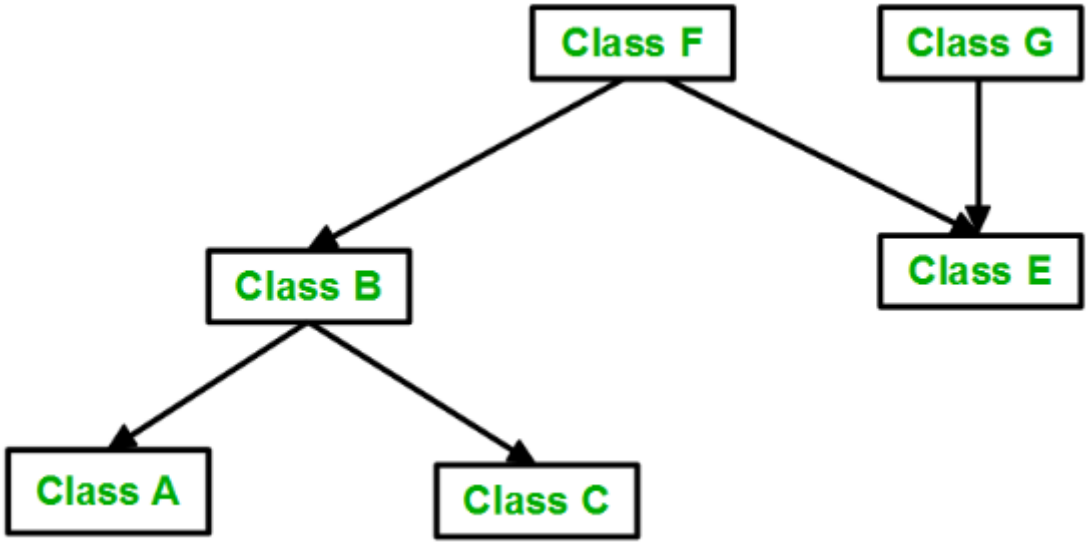
In [21]:
```
# Python program to demonstrate
# Hierarchical inheritance


# 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()
```

```
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

5) Hybrid Inheritance:

Inheritance consisting of multiple types of inheritance is called hybrid inheritance.



In [22]:
```
# Python program to demonstrate
# hybrid inheritance


class School:
    def func1(self):
        print("This function is in school.")


class Student1(School):
    def func2(self):
        print("This function is in student 1. ")


class Student2(School):
    def func3(self):
        print("This function is in student 2.")
```

```
class Student3(Student1, School):
    def func4(self):
        print("This function is in student 3.")


# Driver's code
object = Student3()
object.func1()
object.func2()
```

```
This function is in school.
This function is in student 1.
```