Table of Contents

Inheritance 2

Super Method 8

Class Method 10

@property decorator....................................... 10

Chapter 13

Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class. Parent class is the class being inherited from, also called base class. Child class is the class that inherits from another class, also called derived class.

Ex.

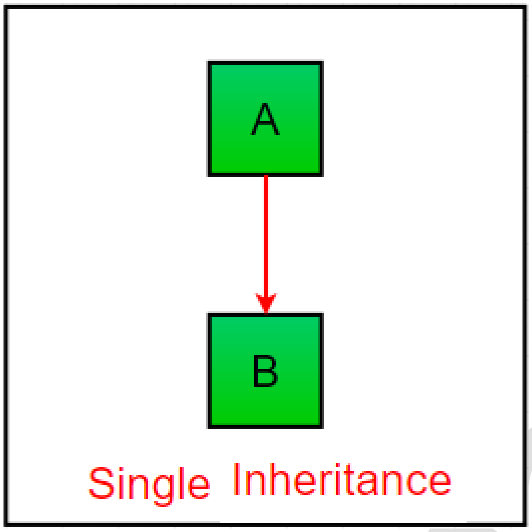
```
class Employee:
    company = "Google"
    def showDetails(self):
        print("This is an employee")

class Programmer(Employee):
    language = "Python"
    def getLanguage(self):
        print(f"The language is {self.language}")
    # overiding
    def showDetails(self):
        print("This is a programmer")

e = Employee()
e.showDetails()
p= Programmer()
p.showDetails()
print(p.company)
```

Types of inheritance:

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

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.



Ex.

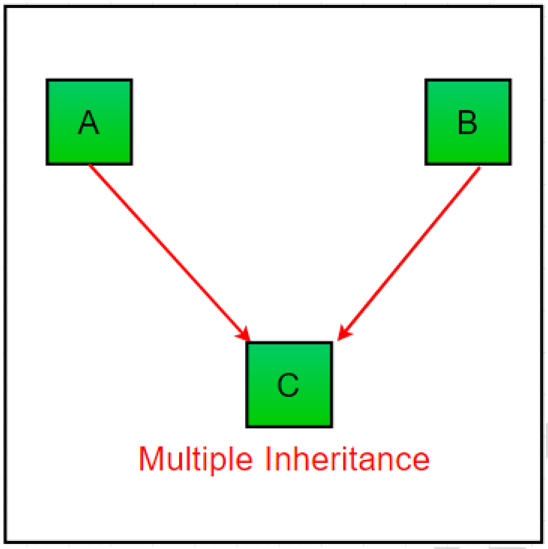
```
# 1.Single Level

# Base class
class ParentSing:
    def func1(self):
        print("This function is in parent class of single level inheritance.")

# Derived class
class ChildSing(ParentSing):
    def func2(self):
        print("This function is in child class of single level inheritance.")

# Driver's code
objectSing = ChildSing()
objectSing.func1()
objectSing.func2()
```

Output:

```
This function is in parent class of single level inheritance.
This function is in child class of single level inheritance.
```

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.



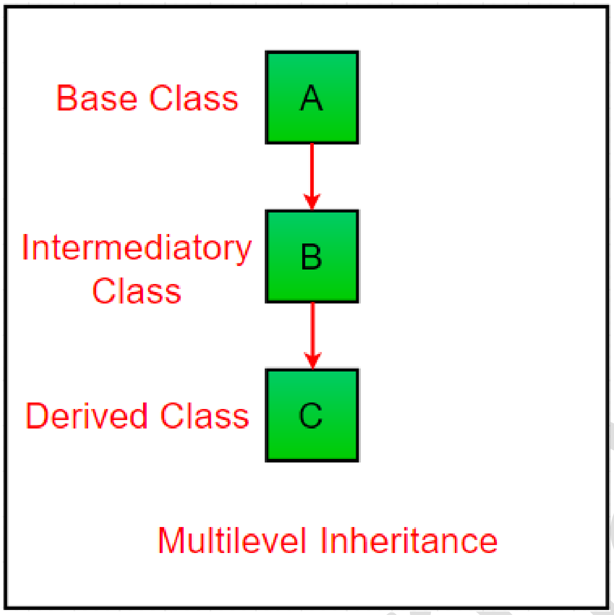
Ex.

```
# 2.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 = "Adam"
s1.mothername = "Eve"
s1.parents()
```

```
Output:
Father: Adam
Mother: Eve
```

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.



Ex.

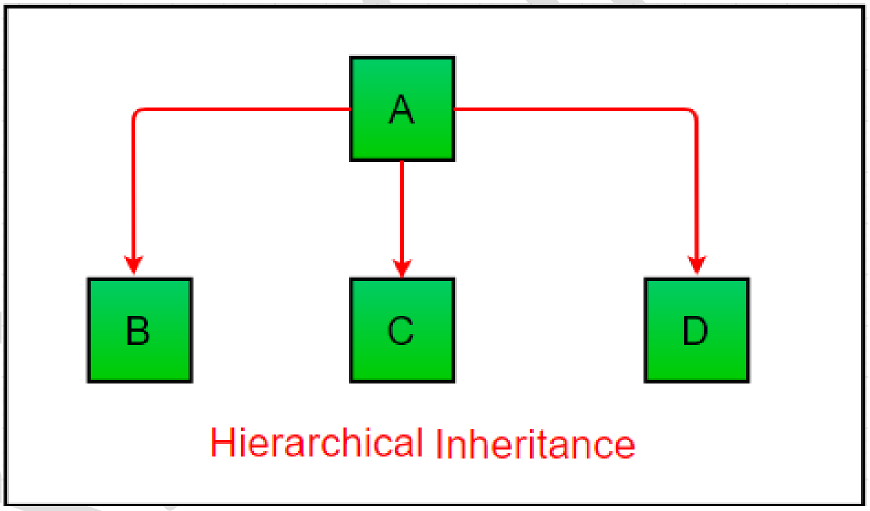
```
# 3.Multilevel inheritance

# 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('sonM', 'fatherM', 'grandFM')
print(s1.grandfathername)
s1.print_name()
```

Output:

```
Grandfather name : grandFM
Father name : fatherM
Son name : sonM
```

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.



Ex.

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

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

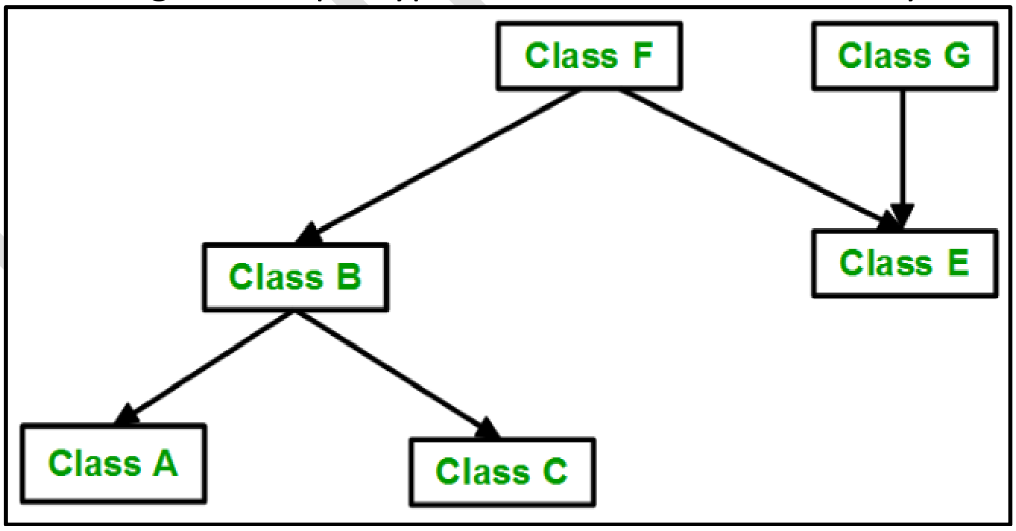
```
# Derivied class2
class Child2(ParentH):
  def func3(self):
  print("This function is in child 2.")


# Driver's code
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

Output:

```
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.



Ex.

```
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()
```

Output:

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

Super Method

Python also has a super() function that will make the child class inherit all the methods and properties from its parent.

Ex1.

```
class per:
    def printer(self):
        print("I am from super Ex parent class")

class chSup(per):
    def runnerChSup(self):
        super().printer()
        print("I am from super Ex child class")

sup = chSup()
sup.runnerChSup()
# I am from super Ex parent class
# I am from super Ex child class
# Ex2.
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)

x = Student("Omar", "Madi")
x.printname()
# Omar Madi

# Ex3.
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019

x = Student("Omar", "Madi")

print(x.firstname)
print(x.lastname)
print(x.graduationyear)
# Omar
# Madi
# 2019
```

Class Method:

```
class employee1:
    company="google"
    salary = "100k"
    location = "delhi"

    # This will not change the employee1.salary
    # def changeSalary(self,sal):
    #     self.salary=sal

    # This will change the employee1.salary
    # Also known as dunder class
    # def changeSalary(self,sal):
    #     self.__class__.salary=sal

    # Does the same as above
    @classmethod
    def changeSalary(cls,sal):
        cls.salary=sal

e = employee1()
e.changeSalary("500k")
print(e.salary)
print(employee1.salary)
```

@property decorator

@property decorator is a built-in decorator in Python which is helpful in defining the properties effortlessly without manually calling the inbuilt function property(). Which is used to return the property attributes of a class from the stated getter, setter and deleter as parameters.

Ex.

```
# Property Decorater
# make a property as a function but let it be property
class employee2:
    company =  "Bharat Gas"
    salary =  1000
    salaryBonus =  2765
    #totalSalary = 3765
    @property # known as getter method
    def totalSalary(self):
        return self.salary + self.salaryBonus

    @totalSalary.setter
    def totalSalary(self,val):
        self.salaryBonus = val - self.salary

e = employee2()
print(e.totalSalary)
e.totalSalary=98765
print(e.salary)
print(e.salaryBonus)
```