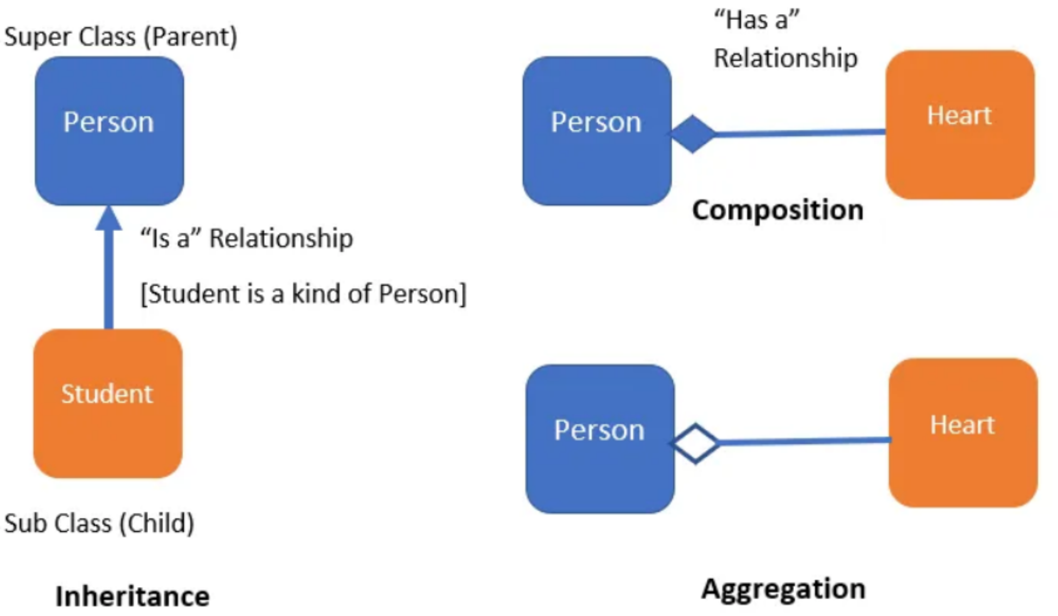
oop part 2

Class Relationships

- Aggregation
- Inheritance
- Composition



Composition

- In composition one class acts as a container of the other class (contents). If you destroy the container there is no existence of contents. That means if the container class creates an object or hold an object of contents.
- Composition established “has-a” relationship between objects. In below code, you can see that the class person is creating a heart object. So, person is the owner of the heart object. We can also say that Person and Heart objects are tightly coupled.

compossition Example

In [1]:

```python
class Heart:
    def __init__(self, heartValves):
        self.heartValves = heartValves

    def display(self):
        return self.heartValves

class Person:
    def __init__(self, fname, lname, address, heartValves):
        self.fname = fname
        self.lname = lname
        self.address = address
        self.heartValves = heartValves
        self.heartObject = Heart(self.heartValves)   # Composition

    def display(self):
        print("First Name: ", self.fname)
        print("Last Name: ", self.lname)
        print("Address: ", self.address)
        print("No of Heart Valves: ", self.heartObject.display())


p = Person("Adam", "syn", "876 Zyx Ln", 4)
p.display()
```

```
First Name:  Adam
Last Name:  syn
Address:  876 Zyx Ln
No of Heart Valves:  4
```

Aggregation

- Not to confuse, aggregation is a form of composition where objects are loosely coupled. There are not any objects or classes owns another object. It just creates a reference. It means if you destroy the container, the content still exists.
- In below code, Person just reference to Heart. There is no tight coupling between Heart and Person object

In [3]:

```python
class Heart:
    def __init__(self, heartValves):
        self.heartValves = heartValves

    def display(self):
        return self.heartValves

class Person:
    def __init__(self, fname, lname, address, heartValves):
        self.fname = fname
        self.lname = lname
        self.address = address
        self.heartValves = heartValves  # Aggregation

    def display(self):
        print("First Name: ", self.fname)
        print("Last Name: ", self.lname)
        print("Address: ", self.address)
        print("No of Healthy Valves: ", hv.display())

hv = Heart(4)
p = Person("Adam", "Lee", "555 wso blvd", hv)
p.display()
```

```
First Name:  Adam
Last Name:  Lee
Address:  555 wso blvd
No of Healthy Valves:  4
```

In [6]:

```
# example
class Customer:

  def __init__(self,name,gender,address):
    self.name = name
    self.gender = gender
    self.address = address

  def print_address(self):
    print(self.address._Address__city,self.address.pin,self.address.state)

  def edit_profile(self,new_name,new_city,new_pin,new_state):
    self.name = new_name
    self.address.edit_address(new_city,new_pin,new_state)

class Address:

  def __init__(self,city,pin,state):
      self.__city = city
      self.pin = pin
      self.state = state

  def get_city(self):
    return self.__city

  def edit_address(self,new_city,new_pin,new_state):
    self.__city = new_city
    self.pin = new_pin
    self.state = new_state

add1 = Address('gandhinagar',382041,'gujarat')
cust = Customer('vishal','male',add1)

cust.print_address()

cust.edit_profile('vishal','mumbai',111111,'maharastra')
cust.print_address()
```

```
gandhinagar 382041 gujarat
mumbai 111111 maharastra
```

In [8]:

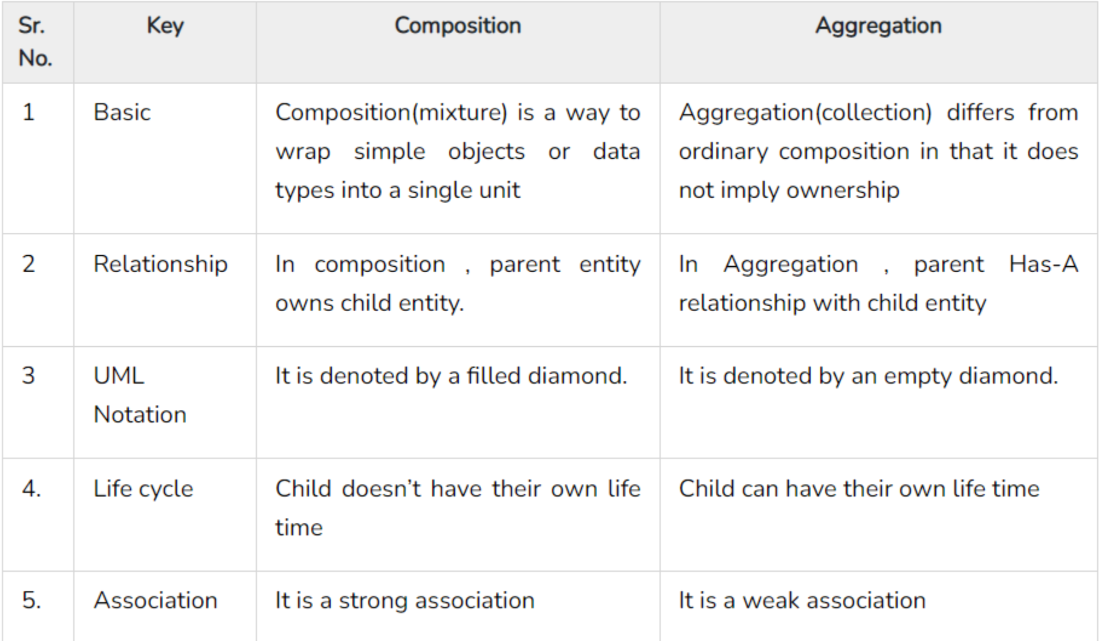
```
# example
class Customer:

  def __init__(self,name,gender,address):
    self.name = name
    self.gender = gender
    self.address = address

  def print_address(self):
    print(self.address.get_city(),self.address.pin,self.address.state)

  def edit_profile(self,new_name,new_city,new_pin,new_state):
    self.name = new_name
    self.address.edit_address(new_city,new_pin,new_state)

class Address:

  def __init__(self,city,pin,state):
      self.__city = city
      self.pin = pin
      self.state = state

  def get_city(self):
    return self.__city

  def edit_address(self,new_city,new_pin,new_state):
    self.__city = new_city
    self.pin = new_pin
    self.state = new_state

add1 = Address('gandhinagar',382041,'gujarat')
cust = Customer('vishal','male',add1)

cust.print_address()

cust.edit_profile('kavit','mumbai',111111,'maharastra')
cust.print_address()
```

```
gandhinagar 382041 gujarat
mumbai 111111 maharastra
```

Aggregation class diagram

- Composition and aggregation are two types of association which is used to represent relationships between two classes.

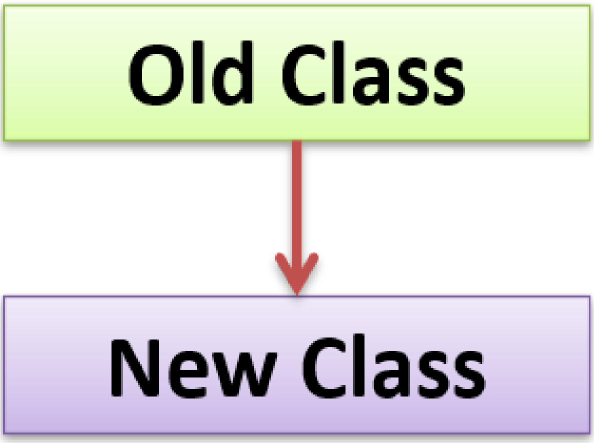
- In Aggregation , parent and child entity maintain Has-A relationship but both can also exist independently. We can use parent and child entity independently. Any modification in the parent entity will not impact the child entity or vice versa. In the above diagram, aggregation is denoted by an empty diamond, which shows their obvious difference in terms of strength of the relationship.
- In Composition, parent owns child entity so child entity can’t exist without parent entity. We can’t directly or independently access child entity. In the UML diagram, composition is denoted by a filled diamond.



Sr. No.	Key	Composition	Aggregation
1	Basic	Composition(mixture) is a way to wrap simple objects or data types into a single unit	Aggregation(collection) differs from ordinary composition in that it does not imply ownership
2	Relationship	In composition , parent entity owns child entity.	In Aggregation , parent Has-A relationship with child entity
3	UML Notation	It is denoted by a filled diamond.	It is denoted by an empty diamond.
4.	Life cycle	Child doesn’t have their own life time	Child can have their own life time
5.	Association	It is a strong association	It is a weak association

Inheritance

- The mechanism of deriving a new class from an old one (existing class) such that the new class inherit all the members (variables and methods) of old class is called inheritance or derivation.



- All classes in python are built from a single super class called ‘object’ so whenever we create a class in python, object will become super class for them internally.

class Mobile(object):

class Mobile:

- The main advantage of inheritance is code reusability.

Declaration of Child Class

class ChildClassName (ParentClassName) :

```
    members of Child class
```

class Mobile (object) :

```
    members of Child class
```

class Mobile :

```
    members of Child class
```

In [16]:

```python
class Father: # Parent Class
    money = 1000
    def show(self):
        print("Parent Class Instance Method")
    @classmethod
    def showmoney(cls):
        print("Parent Class Class Method:", cls.money)
    @staticmethod
    def stat():
        a =10

        print("Parent Class Static Method:", a)
class Son(Father): # Child Class
    def disp(self):
        print("Child Class Instance Method")

s = Son()
s.disp()
s.show()
s.showmoney()
s.stat()
print(s.money)
```

```
Child Class Instance Method
Parent Class Instance Method
Parent Class Class Method: 1000
Parent Class Static Method: 10
1000
```

In [24]:

```python
# Example

# parent
class User:

  def __init__(self):
    self.name = 'vishal'
    self.gender = 'male'

  def login(self):
    print('login')

# child
class Student(User):

  def __init__(self):
    self.rollno = 100

  def enroll(self):
    print('enroll into the course')

u = User()
s = Student()
s.login()
s.enroll()
print(s.name)
```

```
100
login
enroll into the course
```

In [25]:

```python
# Example

# parent
class User:

  def __init__(self):
    self_.name = 'vishal'
    self.gender = 'male'

  def login(self):
    print('login')

# child
class Student(User):

  #def __init__(self):
  #  self.rollno = 100

  def enroll(self):
    print('enroll into the course')

u = User()
s = Student()
s.login()
s.enroll()
print(s.name)
```

```
login
enroll into the course
vishal
```

In [28]:

```
# Example protected variable

# parent
class User:

  def __init__(self):
    self._name = 'vishal'
    self.gender = 'male'

  def login(self):
    print('login')

# child
class Student(User):

  #def __init__(self):
  #  self.rollno = 100

  def enroll(self):
    print('enroll into the course')

u = User()
s = Student()
s.login()
s.enroll()
print(s._name)
```

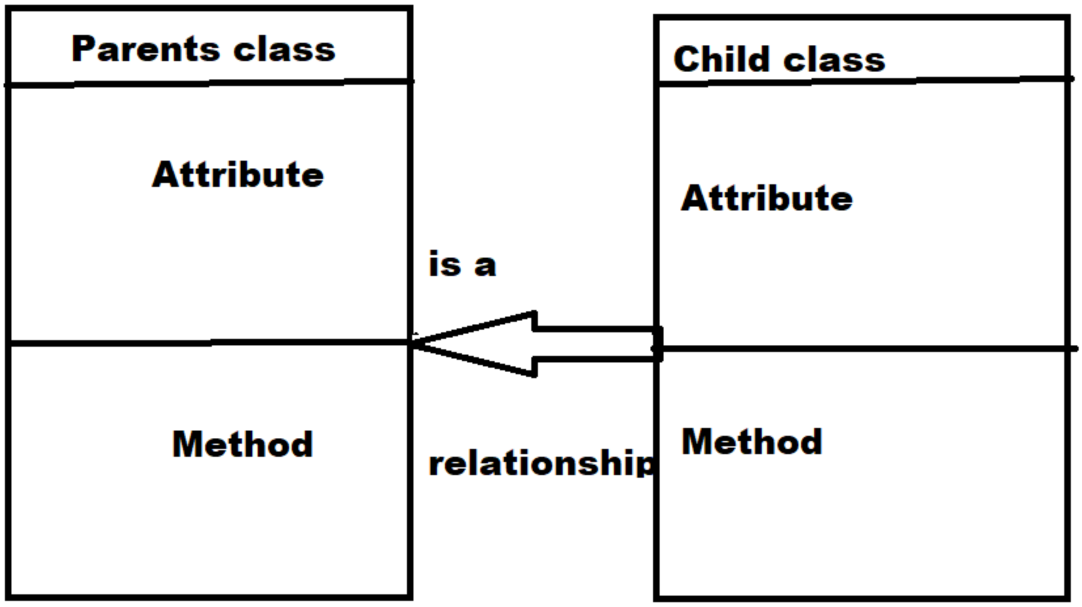
```
login
enroll into the course
vishal
```

In [29]:

```
# Example private variable

# parent
class User:

  def __init__(self):
    self.__name = 'vishal'
    self.gender = 'male'

  def login(self):
    print('login')

# child
class Student(User):

  #def __init__(self):
  #  self.rollno = 100

  def enroll(self):
    print('enroll into the course')

u = User()
s = Student()
s.login()
s.enroll()
print(s._User__name)
```

```
login
enroll into the course
vishal
```

Inheritance Class Diagram



What gets inherited?

- Constructor
- Non Private Attributes
- Non Private Methods

- We can access Parent Class Variables and Methods using Child Class Object
- We can also access Parent Class Variables and Methods using Parent Class Object
- We can not access Child Class Variables and Methods using Parent Class Object

By default, The constructor in the parent class is available to the child class

In [62]:

```python
# Constructor in Inheritance
class Father: # Parent Class
    def __init__(self):
        self.money = 1000
        print("Father Class Constructor")
    def show(self):
        print("Father Class Instance Method")
class Son(Father): # Child Class
    def disp(self):
        print("Son Class Instance Method", self.money)

s = Son()
s.disp()
print("Father Instance Variable:", s.money)
s.show()
```

```
Father Class Constructor
Son Class Instance Method 1000
Father Instance Variable: 1000
Father Class Instance Method
```

In [64]:

```python
# Constructor with Parameter in Inheritance
class Father: # Parent Class
    def __init__(self, m):
        self.money = m
        print("Father Class Constructor")
    def show(self):
        print("Father Class Instance Method")
class Son(Father): # Child Class
    def disp(self):
        print("Son Class Instance Method:", self.money)

s = Son(1000)
s.disp()
print("Father Instance Variable:", s.money)
s.show()
```

```
Father Class Constructor
Son Class Instance Method: 1000
Father Instance Variable: 1000
Father Class Instance Method
```

In [30]:

```python
# constructor example
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    pass

s=SmartPhone(20000, "Apple", 13)
```

```
Inside phone constructor
```

In [32]:

```
s.buy()
```

```
Buying a phone
```

In [31]:

```
b=SmartPhone()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_7044\1847991662.py in <module>
----> 1 b=SmartPhone()

TypeError: __init__() missing 3 required positional arguments: 'price', 'bran
d', and 'camera'
```

Constructor Overriding

- If we write constructor in the both classes, parent class and child class then the parent class constructor is not available to the child class.
- In this case only child class constructor is accessible which means child class constructor is replacing parent class constructor.
- Constructor overriding is used when programmer want to modify the existing behavior of a constructor.

In [33]:

```
# constructor example 2

class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

class SmartPhone(Phone):
    def __init__(self, os, ram):
        self.os = os
        self.ram = ram
        print ("Inside SmartPhone constructor")

s=SmartPhone("Android", 2)
```

```
Inside SmartPhone constructor
```

In [34]:

```
s.brand
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_7044\4068866006.py in <module>
----> 1 s.brand

AttributeError: 'SmartPhone' object has no attribute 'brand'
```

In [36]:

```
s.os
```

Out[36]:

```
'Android'
```

In [65]:

```python
# Constructor Overriding
class Father: # Parent Class
    def __init__(self):
        self.money = 1000
        print("Father Class Constructor")
    def show(self):
        print("Father Class Instance Method")
class Son(Father): # Child Class
    def __init__(self):
        self.money = 5000
        self.car = 'BMW'
        print("Son Class Constructor")
    def disp(self):
        print("Son Class Instance Method")

s = Son()
print(s.money)
print(s.car)
s.disp()
s.show()
```

```
Son Class Constructor
5000
BMW
Son Class Instance Method
Father Class Instance Method
```

In [66]:

```python
# Constructor Overriding with Parameter
class Father: # Parent Class
    def __init__(self, m):
        self.money = m
        print("Father Class Constructor")
    def show(self):
        print("Father Class Instance Method")
class Son(Father): # Child Class
    def __init__(self, r):
        self.money = r
        self.car = 'BMW'
        print("Son Class Constructor")
    def disp(self):
        print("Son Class Instance Method")

s = Son(2000)
print(s.money)
print(s.car)
s.disp()
s.show()
```

```
Son Class Constructor
2000
BMW
Son Class Instance Method
Father Class Instance Method
```

In [39]:

```
# child can't access private members of the class
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    #getter
    def show(self):
        print (self.__price)

class SmartPhone(Phone):
    def check(self):
        print(self.__price)

s=SmartPhone(20000, "Apple", 13)
print(s.brand)
s.check()
```

```
Inside phone constructor
Apple

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_7044\25066133.py in <module>
     18 s=SmartPhone(20000, "Apple", 13)
     19 print(s.brand)
---> 20 s.check()

~\AppData\Local\Temp\ipykernel_7044\25066133.py in check(self)
     14 class SmartPhone(Phone):
     15     def check(self):
---> 16         print(self.__price)
     17
     18 s=SmartPhone(20000, "Apple", 13)

AttributeError: 'SmartPhone' object has no attribute '_SmartPhone__price'
```

In [40]:

```
s.show()
```

```
20000
```

In [41]:

```
# child can't access private members of the class
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    #getter
    def __show(self):
        print (self.__price)

class SmartPhone(Phone):
    def check(self):
        print(self.__price)

s=SmartPhone(20000, "Apple", 13)
print(s.brand)
```

```
Inside phone constructor
Apple
```

In [42]:

```
s.__show()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_7044\1879196203.py in <module>
----> 1 s.__show()

AttributeError: 'SmartPhone' object has no attribute '__show'
```

In [43]:

```
class Parent:

    def __init__(self,num):
        self.__num=num

    def get_num(self):
        return self.__num

class Child(Parent):

    def show(self):
        print("This is in child class")

son=Child(100)
print(son.get_num())
son.show()
```

```
100
This is in child class
```

In [45]:

```python
class Parent:

    def __init__(self,num):
        self.__num=num

    def get_num(self):
        return self.__num

class Child(Parent):

    def __init__(self,val,num):
        self.__val=val

    def get_val(self):
        return self.__val

son=Child(100,10)
print("Child: Val:",son.get_val())
print("Parent: Num:",son.get_num())
```

```
Child: Val: 100

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_7044\4211871849.py in <module>
     17 son=Child(100,10)
     18 print("Child: Val:",son.get_val())
---> 19 print("Parent: Num:",son.get_num())

~\AppData\Local\Temp\ipykernel_7044\4211871849.py in get_num(self)
      5 
      6     def get_num(self):
----> 7         return self.__num
      8 
      9 class Child(Parent):

AttributeError: 'Child' object has no attribute '_Parent__num'
```

In [ ]:

```python

```

In [46]:

```python
class A:
    def __init__(self):
        self.var1=100

    def display1(self,var1):
        print("class A :", self.var1)
class B(A):

    def display2(self,var1):
        print("class B :", self.var1)

obj=B()
obj.display1(200)
```

```
class A : 100
```

In [47]:

```
class A:
    def __init__(self):
        self.var1=100

    def display1(self,var1):
        self.var1=var1
        print("class A :", self.var1)
class B(A):

    def display2(self,var1):
        print("class B :", self.var1)

obj=B()
obj.display1(200)
```

```
class A : 200
```

In [48]:

```
# Method Overriding
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    def buy(self):
        print ("Buying a smartphone")

s=SmartPhone(20000, "Apple", 13)

s.buy()
```

```
Inside phone constructor
Buying a smartphone
```

Super Keyword

- If we write constructor in the both classes, parent class and child class then the parent class constructor is not available to the child class.
- In this case only child class constructor is accessible which means child class constructor is replacing parent class constructor.
- super ( ) method is used to call parent class constructor or methods from the child class.

In [67]:

```python
# Constructor with Super Method
class Father: # Parent Class
    def __init__(self):
        print("Father Class Constructor")
    def show(self):
        print("Father Class Instance Method")
class Son(Father): # Child Class
    def __init__(self):
        super().__init__()      # Calling Parent Class Constructor
        print("Son Class Constructor")
    def disp(self):
        print("Son Class Instance Method")

s = Son()
s.disp()
s.show()
```

```
Father Class Constructor
Son Class Constructor
Son Class Instance Method
Father Class Instance Method
```

In [68]:

```python
# Constructor Parameter with Super Method
class Father: # Parent Class
    def __init__(self, m):
        self.money = m
        print("Father Class Constructor")
    def show(self):
        print("Father Class Instance Method:", self.money)
class Son(Father): # Child Class
    def __init__(self, j, m):
        super().__init__(m)     # Calling Parent Class Constructor
        self.job = j
        print("Son Class Constructor")
    def disp(self):
        print("Son Class Instance Method", self.job)

s = Son('Python', 1000)
s.disp()
s.show()

```

```
Father Class Constructor
Son Class Constructor
Son Class Instance Method Python
Father Class Instance Method: 1000
```

In [61]:

```
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    def buy(self):
        print ("Buying a smartphone")
        # syntax to call parent ka buy method
        super().buy()

s=SmartPhone(20000, "Apple", 13)

s.buy()
```

```
Inside phone constructor
Buying a smartphone
Buying a phone
```

In [51]:

```
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    def buy(self):
        print ("Buying a smartphone")
        # syntax to call parent ka buy method
        super().buy()

s=SmartPhone(20000, "Apple", 13)

s.buy()
```

```
Inside phone constructor
Buying a smartphone
Buying a phone
```

In [52]:

```python
# super -> constuctor
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

class SmartPhone(Phone):
    def __init__(self, price, brand, camera, os, ram):
        print('Inside smartphone constructor')
        super().__init__(price, brand, camera)
        self.os = os
        self.ram = ram
        print ("Inside smartphone constructor")

s=SmartPhone(20000, "Samsung", 12, "Android", 2)

print(s.os)
print(s.brand)
```

```
Inside smartphone constructor
Inside phone constructor
Inside smartphone constructor
Android
Samsung
```

In [53]:

```python
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    def buy(self):
        print ("Buying a smartphone")
        # syntax to call parent ka buy method


s=SmartPhone(20000, "Apple", 13)

s.super().buy()
```

```
Inside phone constructor

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_7044\4051818540.py in <module>
     17 s=SmartPhone(20000, "Apple", 13)
     18 
---> 19 s.super().buy()

AttributeError: 'SmartPhone' object has no attribute 'super'
```

In [54]:

```python
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    def buy(self):
        print ("Buying a smartphone")
        # syntax to call parent ka buy method
        print(super().brand)

s=SmartPhone(20000, "Apple", 13)

s.buy()
```

```
Inside phone constructor
Buying a smartphone

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_7044\3539378570.py in <module>
     17 s=SmartPhone(20000, "Apple", 13)
     18 
---> 19 s.buy()

~\AppData\Local\Temp\ipykernel_7044\3539378570.py in buy(self)
     13         print ("Buying a smartphone")
     14         # syntax to call parent ka buy method
---> 15         print(super().brand)
     16 
     17 s=SmartPhone(20000, "Apple", 13)

AttributeError: 'super' object has no attribute 'brand'
```

Inheritance in summary

- A class can inherit from another class.
- Inheritance improves code reuse
- Constructor, attributes, methods get inherited to the child class
- The parent has no access to the child class
- Private properties of parent are not accessible directly in child class
- Child class can override the attributes or methods. This is called method overriding
- super() is an inbuilt function which is used to invoke the parent class methods and constructor

In [55]:

```python
class Parent:

    def __init__(self,num):
      self.__num=num

    def get_num(self):
      return self.__num

class Child(Parent):

    def __init__(self,num,val):
      super().__init__(num)
      self.__val=val

    def get_val(self):
      return self.__val

son=Child(100,200)
print(son.get_num())
print(son.get_val())
```

```
100
200
```

In [56]:

```python
class Parent:
    def __init__(self):
        self.num=100

class Child(Parent):

    def __init__(self):
        super().__init__()
        self.var=200

    def show(self):
        print(self.num)
        print(self.var)

son=Child()
son.show()
```

```
100
200
```

In [57]:

```python
class Parent:
    def __init__(self):
        self.__num=100

    def show(self):
        print("Parent:",self.__num)

class Child(Parent):
    def __init__(self):
        super().__init__()
        self.__var=10

    def show(self):
        print("Child:",self.__var)

obj=Child()
obj.show()
```

```
Child: 10
```

In [58]:

```python
class Parent:
    def __init__(self):
        self.__num=100

    def show(self):
        print("Parent:",self.__num)

class Child(Parent):
    def __init__(self):
        super().__init__()
        self.__var=10

    def show(self):
        print("Child:",self.__var)

obj=Child()
obj.show()
```

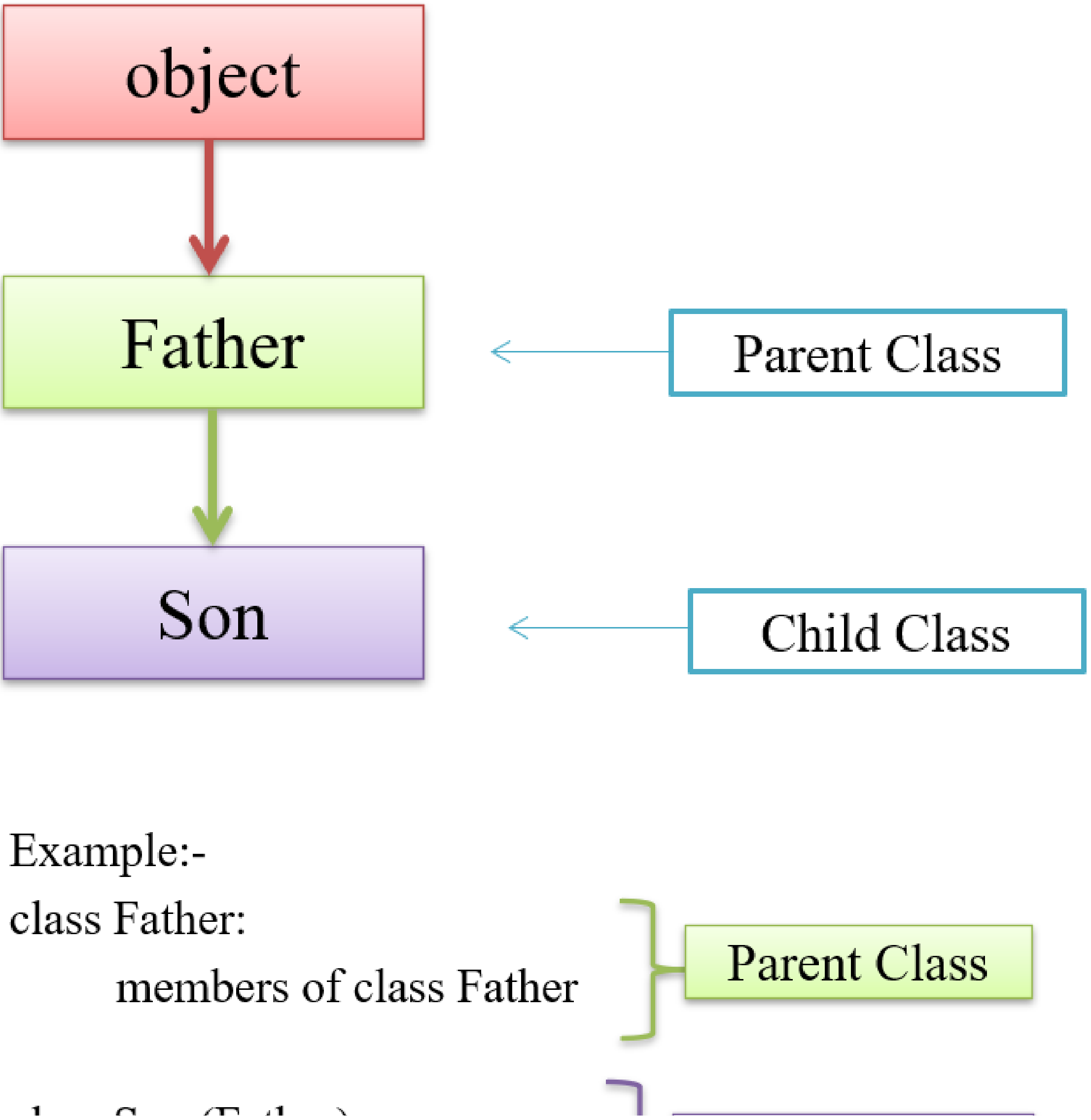
```
Child: 10
```

Types of Inheritance

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance(Diamond Problem)
- Hybrid Inheritance

Single Inheritance

If a class is derived from one base class (Parent Class), it is called Single Inheritance.



Example:-

class Father:

members of class Father

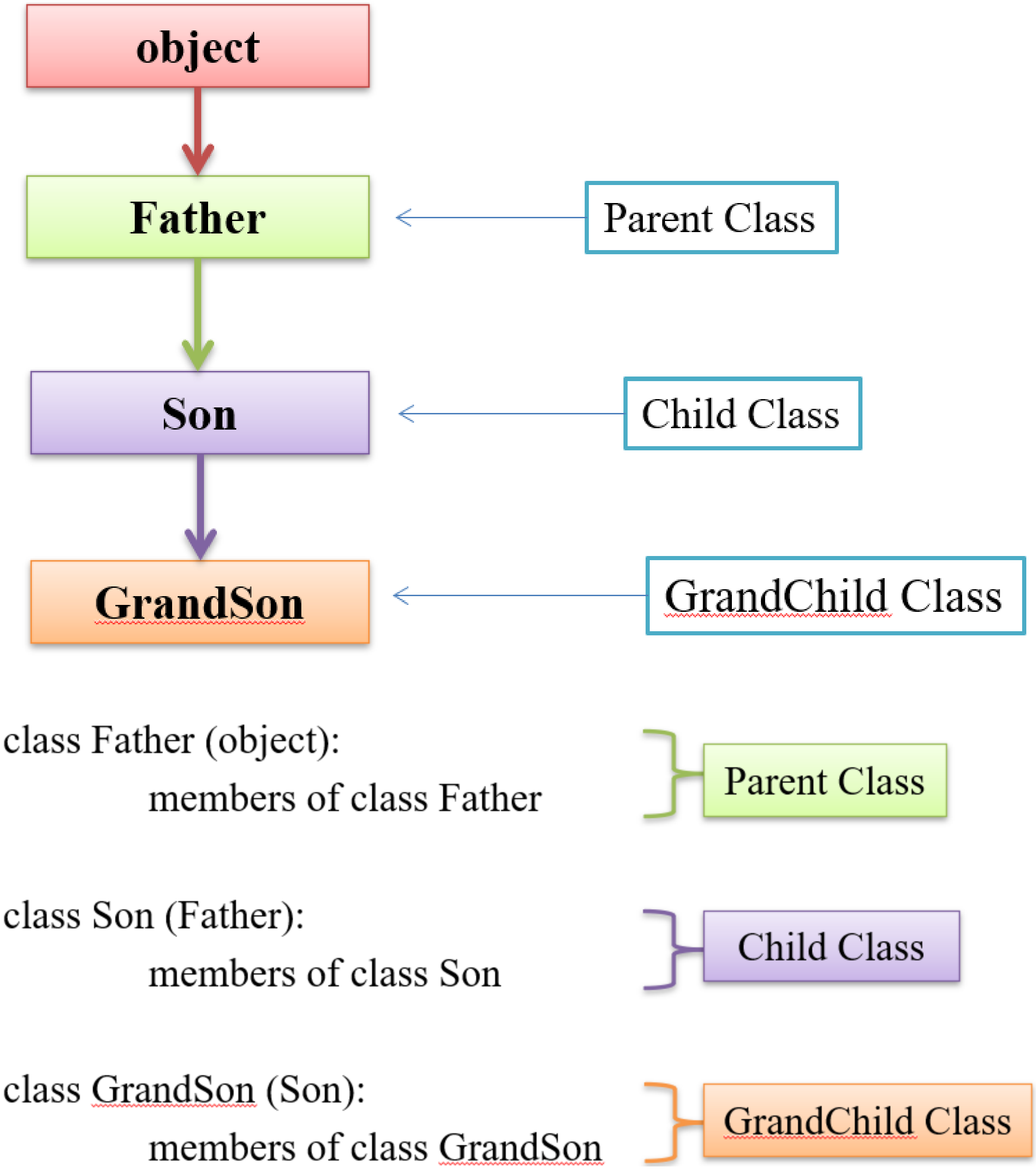
In [60]:

```python
# single inheritance
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    pass

SmartPhone(1000,"Apple","13px").buy()
```

```
Inside phone constructor
Buying a phone
```

Multi-level Inheritance

- In multi-level inheritance, the class inherits the feature of another derived class (Child Class).



```
class Father (object):
        members of class Father

class Son (Father):
        members of class Son

class GrandSon (Son):
        members of class GrandSon
```

In [69]:

```
# Multi-level Inheritance
class Father:
    def __init__(self):
        print("Father Class Constructor")
    def showF(self):
        print("Father Class Method")
class Son(Father):
    def __init__(self):
        print("Son Class Constructor")
    def showS(self):
        print("Son Class Method")
class GrandSon(Son):
    def __init__(self):
        print("GrandSon Class Constructor")
    def showG(self):
        print("GrandSon Class Method")

g = GrandSon()
g.showF()
g.showS()
g.showG()
```

```
GrandSon Class Constructor
Father Class Method
Son Class Method
GrandSon Class Method
```

In [72]:

```
# Multi-level Inheritance
class Father:
    def __init__(self):
        print("Father Class Constructor")
    def showF(self):
        print("Father Class Method")
class Son(Father):
    def __init__(self):
        super().__init__()
        print("Son Class Constructor")
    def showS(self):
        print("Son Class Method")
class GrandSon(Son):
    def __init__(self):
        super().__init__()
        print("GrandSon Class Constructor")
    def showG(self):
        print("GrandSon Class Method")

g = GrandSon()
g.showF()
g.showS()
g.showG()
```

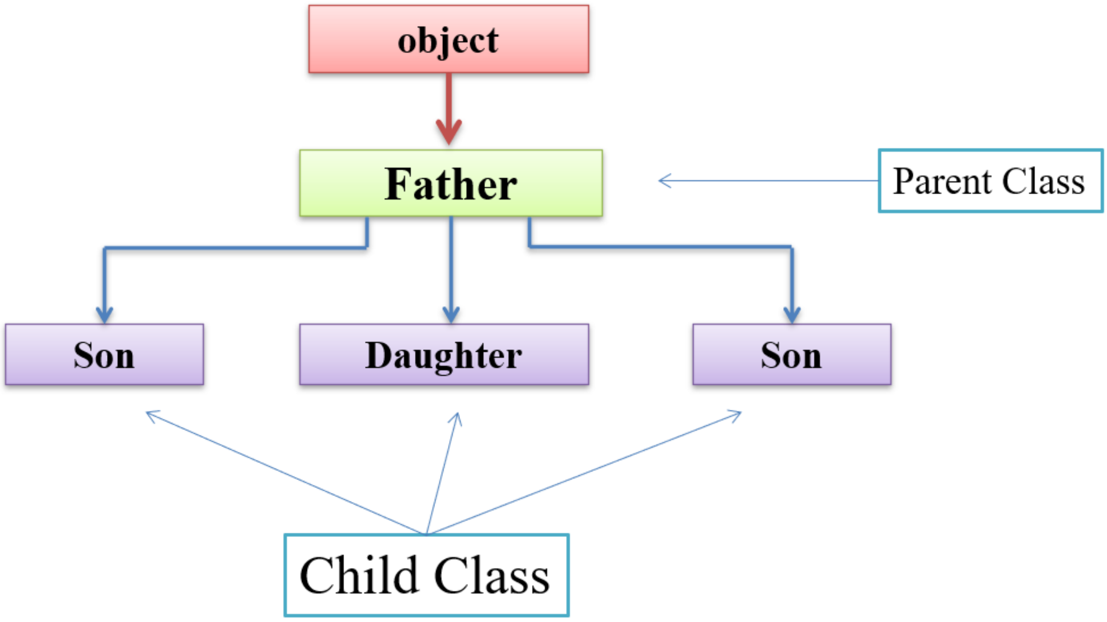
```
Father Class Constructor
Son Class Constructor
GrandSon Class Constructor
Father Class Method
Son Class Method
GrandSon Class Method
```

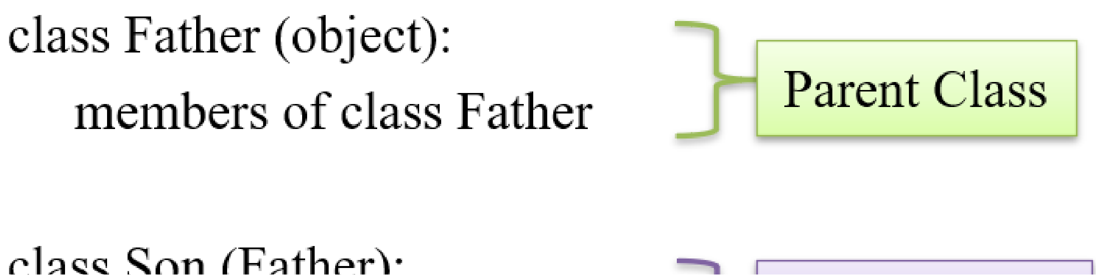
In [71]:

```python
# multilevel
class Product:
    def review(self):
        print ("Product customer review")

class Phone(Product):
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    pass

s=SmartPhone(20000, "Apple", 12)

s.buy()
s.review()
```

```
Inside phone constructor
Buying a phone
Product customer review
```

Hierarchical Inheritance





class Father (object):

members of class Father

class Son (Father):

In [73]:

```
# Hierarchical Inheritance
class Father:
    def __init__(self):
        print("Father Class Constructor")
    def showF(self):
        print("Father Class Method")
class Son(Father):
    def __init__(self):
        print("Son Class Constructor")
    def showS(self):
        print("Son Class Method")
class Daughter(Father):
    def __init__(self):
        print("Daughter Class Constructor")
    def showD(self):
        print("Daughter Class Method")
d = Daughter()
d.showF()
d.showD()
s = Son()
s.showF()
s.showS()
```

```
Daughter Class Constructor
Father Class Method
Daughter Class Method
Son Class Constructor
Father Class Method
Son Class Method
```

In [74]:

```python
# Hierarchical Inheritance
class Father:
    def __init__(self):
        print("Father Class Constructor")
    def showF(self):
        print("Father Class Method")
class Son(Father):
    def __init__(self):
        super().__init__()
        print("Son Class Constructor")
    def showS(self):
        print("Son Class Method")
class Daughter(Father):
    def __init__(self):
        super().__init__()
        print("Daughter Class Constructor")
    def showD(self):
        print("Daughter Class Method")

d = Daughter()
d.showF()
d.showD()
s = Son()
s.showF()
s.showS()
```

```
Father Class Constructor
Daughter Class Constructor
Father Class Method
Daughter Class Method
Father Class Constructor
Son Class Constructor
Father Class Method
Son Class Method
```

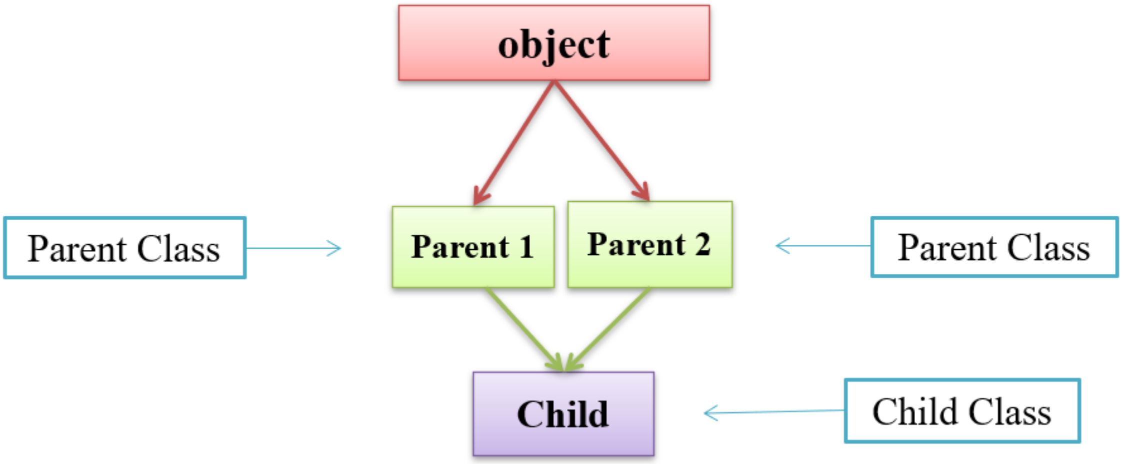
In [75]:

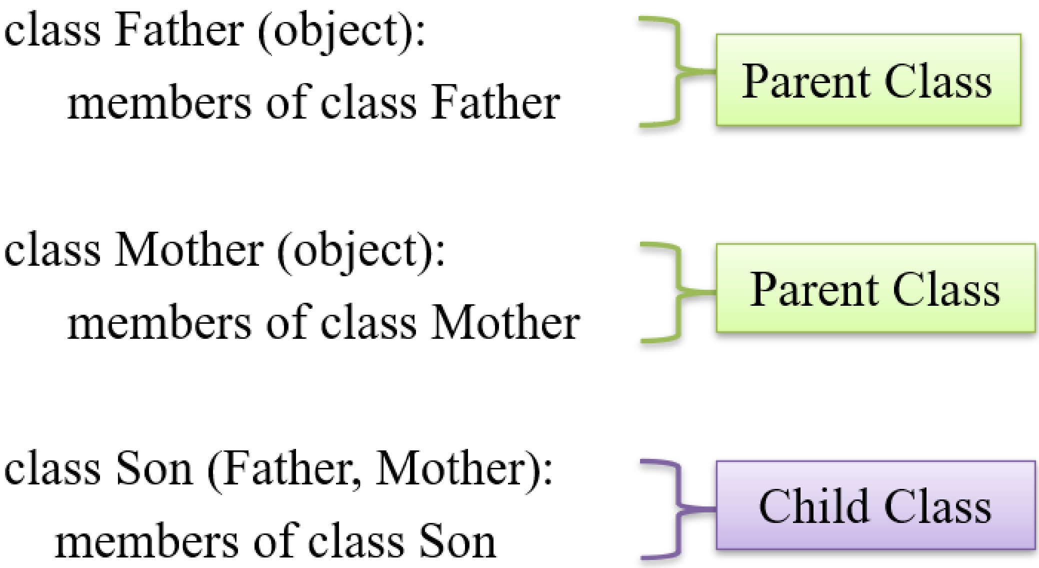
```python
# Hierarchical
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    pass

class FeaturePhone(Phone):
    pass

SmartPhone(1000,"Apple","13px").buy()
FeaturePhone(10,"Lava","1px").buy()
```

```
Inside phone constructor
Buying a phone
Inside phone constructor
Buying a phone
```

Multiple Inheritance

- If a class is derived from more than one parent class, then it is called multiple inheritance.





```
class Father (object):
     members of class Father

class Mother (object):
     members of class Mother

class Son (Father, Mother):
    members of class Son
```

In [76]:

```python
# Multiple Inheritance
class Father:
    def __init__(self):
        print("Father Class Constructor")
    def showF(self):
        print("Father Class Method")

class Mother:
    def __init__(self):
        print("Mother Class Constructor")
    def showM(self):
        print("Mother Class Method")

class Son(Father, Mother):
    def __init__(self):
        print("Son Class Constructor")
    def showS(self):
        print("Son Class Method")

s = Son()
s.showF()
s.showM()
s.showS()
```

```
Son Class Constructor
Father Class Method
Mother Class Method
Son Class Method
```

In [77]:

```python
# Multiple Inheritance
class Father:
    def __init__(self):
        print("Father Class Constructor")
    def showF(self):
        print("Father Class Method")

class Mother:
    def __init__(self):
        print("Mother Class Constructor")
    def showM(self):
        print("Mother Class Method")

class Son(Father, Mother):
    def __init__(self):
        super().__init__() # Calling Parent Class Constructor
        print("Son Class Constructor")
    def showS(self):
        print("Son Class Method")

s = Son()
s.showF()
s.showM()
s.showS()

```

```
Father Class Constructor
Son Class Constructor
Father Class Method
Mother Class Method
Son Class Method
```

In [78]:

```python
# Multiple
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class Product:
    def review(self):
        print ("Customer review")

class SmartPhone(Phone, Product):
    pass

s=SmartPhone(20000, "Apple", 12)

s.buy()
s.review()
```

```
Inside phone constructor
Buying a phone
Customer review
```

In [80]:

```python
# Multiple Inheritance
class Father:
    def __init__(self):
        super().__init__()                  # Calling Parent Class Construct
        print("Father Class Constructor")
    def showF(self):
        print("Father Class Method")
class Mother:
    def __init__(self):
        super().__init__()                  # Calling Parent Class Construct
        print("Mother Class Constructor")
    def showM(self):
        print("Mother Class Method")

class Son(Father, Mother):
    def __init__(self):
        super().__init__()                  # Calling Parent Class Construct
        print("Son Class Constructor")
    def showS(self):
        print("Son Class Method")

s = Son()
s.showF()
s.showM()
s.showS()

```

```
Mother Class Constructor
Father Class Constructor
Son Class Constructor
Father Class Method
Mother Class Method
Son Class Method
```

In [81]:

```python
class A:

    def m1(self):
        return 20

class B(A):

    def m1(self):
        return 30

    def m2(self):
        return 40

class C(B):

    def m2(self):
        return 20
obj1=A()
obj2=B()
obj3=C()
print(obj1.m1() + obj3.m1()+ obj3.m2())
```

```
70
```

In [ ]:

```python
class A:

    def m1(self):
        return 20

class B(A):

    def m1(self):
        val=super().m1()+30
        return val

class C(B):

    def m1(self):
        val=self.m1()+20
        return val
obj=C()
print(obj.m1())
```

Method Resolution Order (MRO)

-n In multiple inheritance scenarios, any specific attribute or method will initially be searched in the current class. If not found in the current class, then next search continues into parent classes in depth-first left to right fashion. Searching in this order is called Method Resolution Order (MRO).

Three principles of MRO:

- The first principle is to search for the subclass before going for its base classes. If class B is inherited from A, it will search B first and then goes to A.
- The second principle is, if any class is inherited from several classes, it searches in the order from left to right in the base classes. For example, if class C is inherited from A and B, syntactically class C(A, B), then first it will search in A and then in B.
- The third principle is that it will not visit any class more than once. That means a class in the inheritance hierarchy is traversed only once exactly. Understanding MRO gives you clear idea

regarding which classes are being executed and in which sequence. We have a predefined method to see the sequence of execution of classes. It is: classname.mro()

Method Resolution Order (MRO)

- In the multiple inheritance scenario members of class are searched first in the current class. If not found, the search continues into parent classes in depth-first, left to right manner without searching the same class twice.
- Search for the child class before going to its parent class.
- When a class is inherited from several classes, it searches in the order from left to right in the parent classes.
- It will not visit any class more than once which means a class in the inheritance hierarchy is traversed only once exactly.

```
s = Son()
```

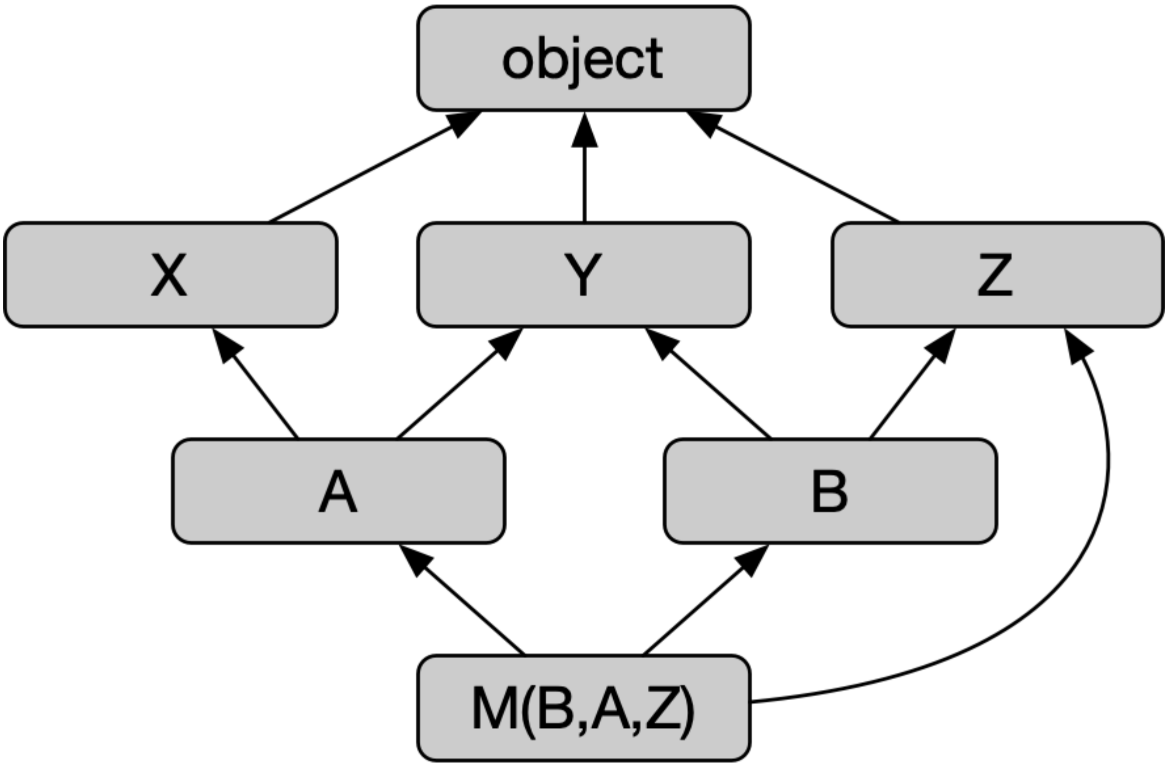
- The search will start from Son. As the object of Son is created, the constructor of Son is called.
- Son has super().**init**() inside his constructor so its parent class, the one in the left side ‘Father’ class’s constructor is called.
- Father class also has super().**init**() inside his constructor so its parent ‘object’ class’s constructor is called.
- Object does not have any constructor so the search will continue down to right hand side class (Mother) of object class so Mother class’s constructor is called.
- As Mother class also has super().**inti**() so its parent class ‘object’ constructor is called but as object class already visited, the search will stop here

For instance, what’s search sequence of class M?

In [1]:

```
class X:pass
class Y: pass
class Z:pass
class A(X,Y):pass
class B(Y,Z):pass
class M(B,A,Z):pass
print(M.mro())
```

```
[<class '__main__.M'>, <class '__main__.B'>, <class '__main__.A'>, <class '__ma
in__.X'>, <class '__main__.Y'>, <class '__main__.Z'>, <class 'object'>]
```



C3 Algorithm

C3 follows these two equation:

L[object] = [object]

L[C(B1…BN)] = [C] + merge(L[B1]…L[BN], [B1, … ,BN])

L[C] is the MRO of class C, it will evaluate to a list.

- The key process is merge, it get a list and generate a list by this way:

1. First, check the first list's head element(L[B1]) as H.
2. If H is not in the tail of other list, output it, and remove it from all of the list, then go to step 1. Otherwise, check the next list's head as H, go to step 2. (tail means the rest of the list except the first element)
3. If merge's list is empty, end algorithm. If list is not empty but not able to find element to output, raise error.

That seems complicated, I'll use the previous example again to explain the calculation of C3.

Let's begin with the easy ones. Firstly, calculate A's MRO:

Let's begin with the easy ones. Firstly, calculate A's MRO:

L[A(X,Y)]=[A]+merge(L[X],L[Y],[X,Y])

```
 =[A]+merge([X,obj],[Y,obj],[X,Y])

 # X is not tail of other list, use it as H

=[A,X]+merge([obj],[Y,obj],[Y])

# obj is in the tail of[Y.obj], use Y as H

=[A,X,Y]+merge([obj],[obj]]

 =[A,X,Y,obj]
```

B’s MRO [B,Y,Z,obj] and Z’s MRO [z,obj] can also be calculated.

Now we can get M’s MRO:

L[M(B,A,Z)]=[M]+merge(L[B],L[A],L[Z],[B,A,Z])

```
=[M]+merge([B,Y,Z,obj],[A,X,Y,obj],[Z,obj],[B,A,Z])

=[M,B]+merge([Y,Z,obj],[A,X,Y,obj],[Z,obj],[A,Z])

# Y is in the tail of [A,X,Y,obj], use A as H

=[M,B,A]+merge([Y,Z,obj],[X,Y,obj],[Z,obj],[Z])

# Y is in the tail of [X,Y,obj], use X as H

=[M,B,A,X]+merge([Y,Z,obj],[Y,obj],[Z,obj],[Z])
```

MRO and super()

- For instance, C’s MRO is C,A,B,Base,obj, so after enter A, it will output enter B rather than enter base.

In [2]:

```python
class Base:
    def __init__(self):
        print('enter base')
        print('leave base')


class A(Base):
    def __init__(self):
        print('enter A')
        super(A, self).__init__()
        print('leave A')


class B(Base):
    def __init__(self):
        print('enter B')
        super(B, self).__init__()
        print('leave B')


class C(A, B):
    def __init__(self):
        print('enter C')
        super(C, self).__init__()
        print('leave C')

c = C()
```

```
enter C
enter A
enter B
enter base
leave base
leave B
leave A
leave C
```

Example

In [4]:

```python
class A:
    def myname(self):
        print("I am a class A")

class B(A):
    def myname(self):
        print("I am a class B")

class C(A):
    def myname(self):
        print("I am a class C")
c = C()
print(c.myname())
print(C.mro())
```

```
I am a class C
None
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

In [ ]:

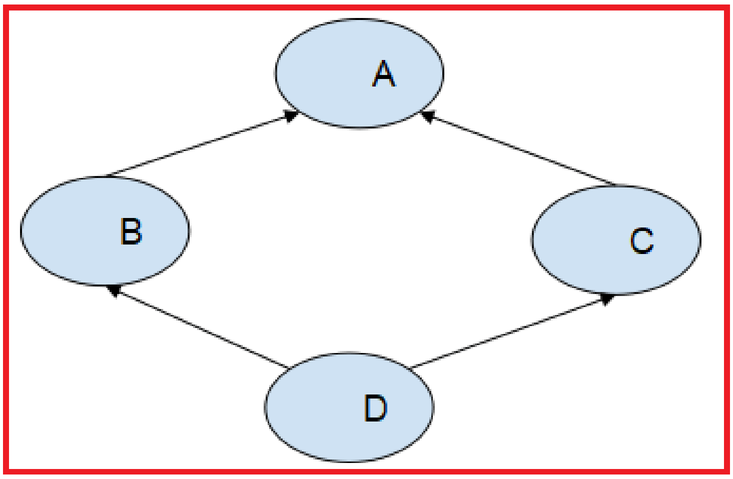
```

```

In [3]:

```
class A:
    def myname(self):
        print(" I am a class A")
class B(A):
    def myname(self):
        print(" I am a class B")
class C(A):
    def myname(self):
        print("I am a class C")

# classes ordering
class D(B, C):
    pass
d = D()
d.myname()
print(D.mro())
```

```
 I am a class B
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__ma
in__.A'>, <class 'object'>]
```

Note: Object is a default super class in python.



- mro(A)=A, object
- mro(B)=B, A, object
- mro(C)=C, A, object
- mro(D)=D, B, C, A, object Note: Object is a default super class in python.

In [11]:

```
class A:
   def m1(self):
       print("m1 from A")
class B(A):
   def m1(self):
       print("m1 from B")
class C(A):
   def m1(self):
       print("m1 from C")
class D(B, C):
   def m1(self):
       print("m1 from D")
print(A.mro())
print(B.mro())
print(C.mro())
print(D.mro())
```

```
[<class '__main__.A'>, <class 'object'>]
[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__ma
in__.A'>, <class 'object'>]
```

In [12]:

```
class A:
   def m1(self):
       print("m1 from A")
class B(A):
   def m1(self):
       print("m1 from B")
class C(A):
   def m1(self):
       print("m1 from C")
class D(B, C):
   def m1(self):
       print("m1 from D")
c=C()
c.m1()
print(C.mro())
```

```
m1 from C
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

In [13]:

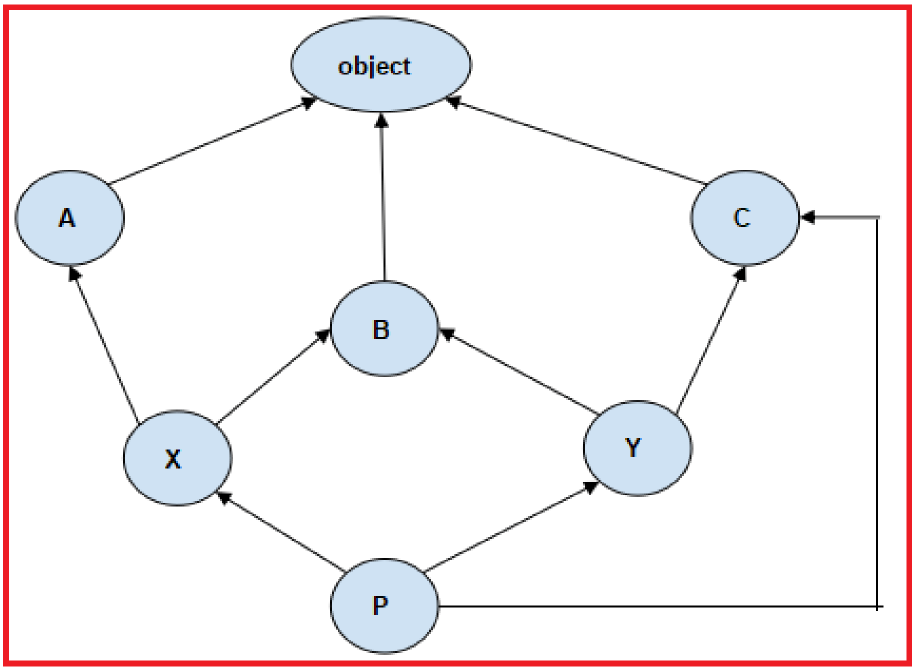
```
class A:
   def m1(self):
       print("m1 from A")
class B(A):
   def m1(self):
       print("m1 from B")
class C(A):
   def m2(self):
       print("m2 from C")
class D(B, C):
   def m1(self):
       print("m1 from D")
c=C()
c.m1()
print(C.mro())
```

```
m1 from A
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

In [14]:

```
class A:
    def m1(self):
        print("m1 from A")
class B(A):
    def m1(self):
        print("m1 from B")
class C(A):
    def m1(self):
        print("m1 from C")
class D(B, C):
    def m1(self):
        print("m1 from D")
d=D()
d.m1()
print(D.mro())
```

```
m1 from D
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__ma
in__.A'>, <class 'object'>]
```



- mro(A)=A, object
- mro(B)=B, object
- mro(C)=C, object
- mro(X)=X, A, B, object
- mro(Y)=Y, B, C, object
- mro(P)=P, X, A, Y, B, C, object

In [15]:

```python
class A:
   def m1(self):
       print("m1 from A")
class B:
   def m1(self):
       print("m1 from B")
class C:
   def m1(self):
       print("m1 from C")
class X(A, B):
   def m1(self):
       print("m1 from C")
class Y(B, C):
   def m1(self):
       print("m1 from A")
class P(X, Y, C):
   def m1(self):
       print("m1 from P")
print(A.mro())#AO
print(X.mro())#XABO
print(Y.mro())#YBCO
print(P.mro())#PXAYBCO
```

```
[<class '__main__.A'>, <class 'object'>]
[<class '__main__.X'>, <class '__main__.A'>, <class '__main__.B'>, <class 'obje
ct'>]
[<class '__main__.Y'>, <class '__main__.B'>, <class '__main__.C'>, <class 'obje
ct'>]
[<class '__main__.P'>, <class '__main__.X'>, <class '__main__.A'>, <class '__ma
in__.Y'>, <class '__main__.B'>, <class '__main__.C'>, <class 'object'>]
```

In [5]:

```python
class A:
    def process(self):
        print('A process()')


class B:
    def process(self):
        print('B process()')


class C(A, B):
    def process(self):
        print('C process()')


class D(C,B):
    pass


obj = D()
obj.process()

print(D.mro())
```

```
C process()
[<class '__main__.D'>, <class '__main__.C'>, <class '__main__.A'>, <class '__ma
in__.B'>, <class 'object'>]
```

In [7]:

```
class A:
    def process(self):
        print('A process()')


class B(A):
    pass


class C(A):
    def process(self):
        print('C process()')


class D(B,C):
    pass


obj = D()
print(D.mro())
obj.process()

```

```
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__ma
in__.A'>, <class 'object'>]
C process()
```

In [8]:

```
class A:
    def process(self):
        print('A process()')


class B(A):
    def process(self):
        print('B process()')


class C(A, B):
    pass


obj = C()
print(C.mro())
obj.process()

```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-8f81c48920b9> in <module>
      9
     10
---> 11 class C(A, B):
     12     pass
     13

TypeError: Cannot create a consistent method resolution
order (MRO) for bases A, B
```

- The problem comes from the fact that class A is a super class for both C and B. If you construct MRO then it should be like this:
- C -> A -> B -> A

- Then according to the rule (good head) A should NOT be ahead of B as A is super class of B. So new MRO must be like this:
- C -> B -> A
- But A is also direct super class of C. So, if a method is in both A and B classes then which version should class C call? According to new MRO, the version in B is called first ahead of A and that is not according to inheritance rules (specific to generic) resulting in Python to throw error.