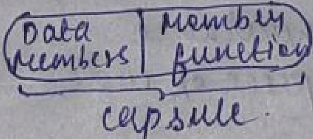
* 4 pillars of OOPs:

① Encapsulation ② Inheritance ③ Polymorphism
④ Abstraction.

① **Encapsulation:** * Wrapping the Data members & member functions inside an entity or a class.

* Basically it is Data Hiding which don't allow the direct access of members of a class.
* Means marking the members under private section.
* **Perfect Encapsulation:** when all the members of a class are under private section which a user can't access it directly.

* A class is an example of encapsulation as all the data members & members functions are wrapped inside a class.



② **Inheritance:** * In simple inheriting the properties & behaviours of base class into child class.

* Reusing the properties & behaviours and extending the existing class is said to Inheritance

i.e Base class | Parent class | super class

↑↓

child class | sub class | derived class

* We need Inheritance for reusability factor we can use this code over again.

i.e Parent class

↓

child class

* syntax of using Inheritance

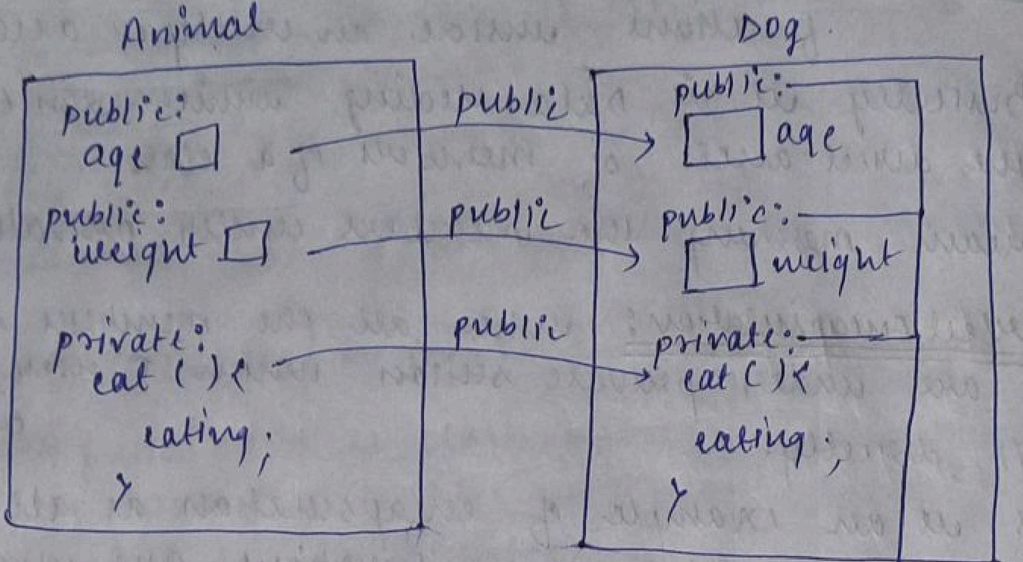
class child : ______ Parent

↓ child class/ Derived class

↓ Mode of inheritance ↳ private, public, protected

↓ Parent class/ Base class.

*



class Dog : public Animal

↓ syntax of inheritance for above piece of code

How does the mode of inheritance define the access modifiers of members of derived/ child class.

* How does the mode of Inheritance define the access modifiers of members of derived/child class?

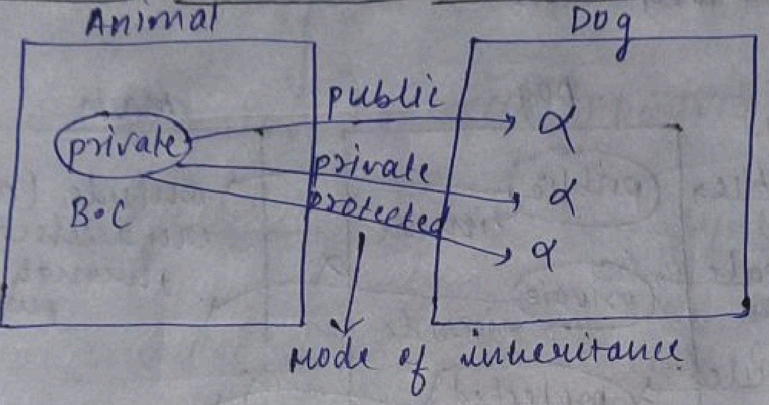
→ To understand we need to recall a chart.

Base class Access Modifier	Mode of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	NA	NA	NA

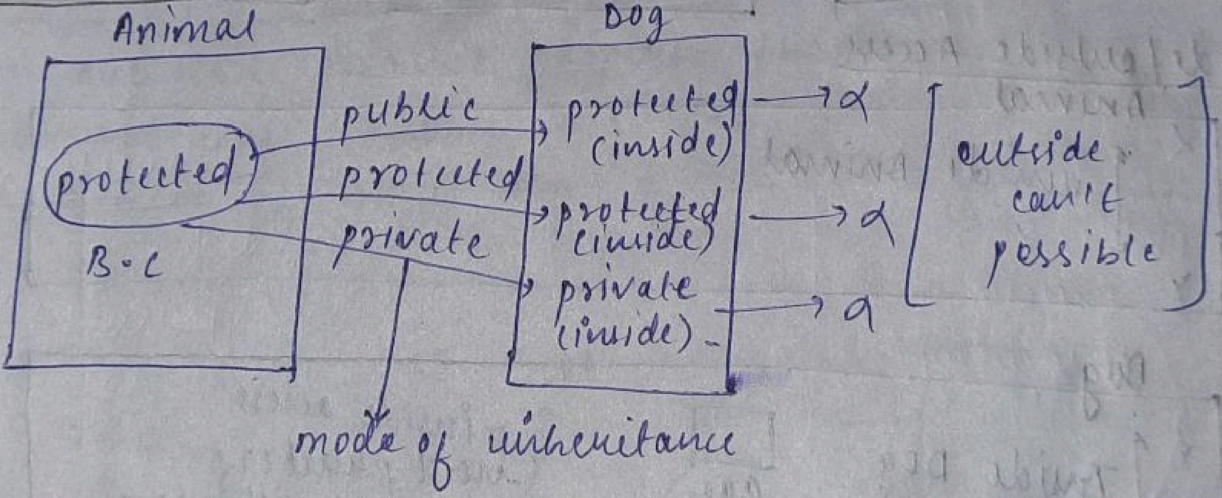
* We know what is private & public access modifiers but what is protected access modifier?

→ Protected : members declare under protected can be access ~~under~~ ~~privat~~ inside of base class and if it is used as a mode of inheritance then it can be access inside the derived class. can't be access outside of class in the both case

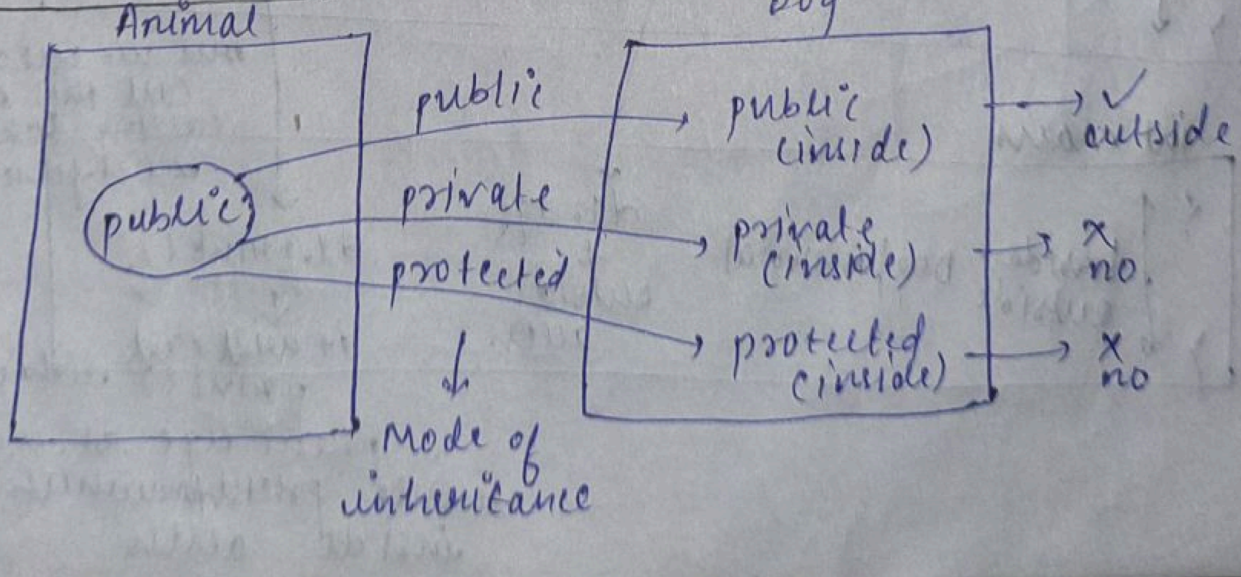
* Private Base class can't be inherited

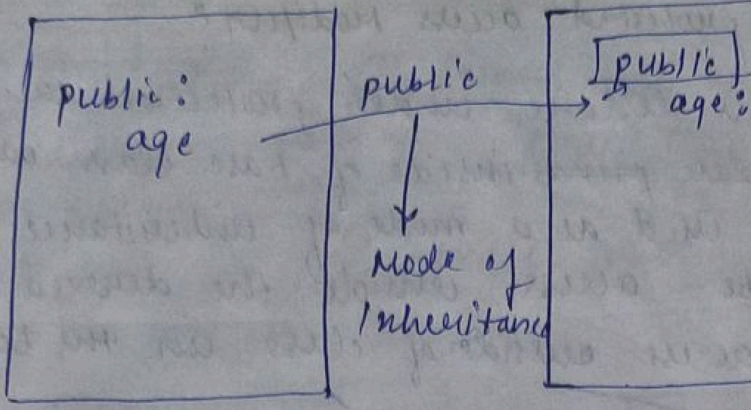


* Protected Base class

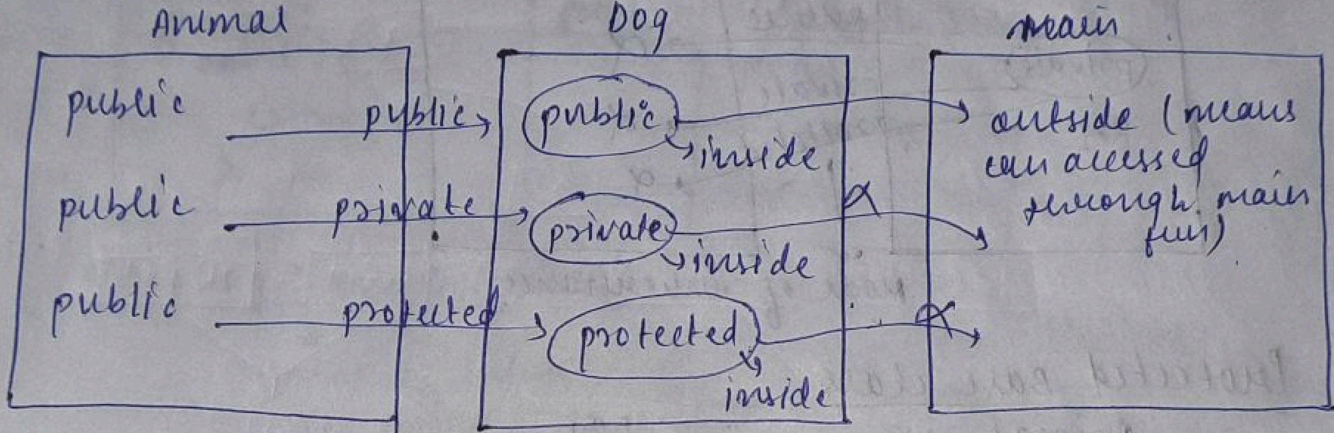


* Public Base class:

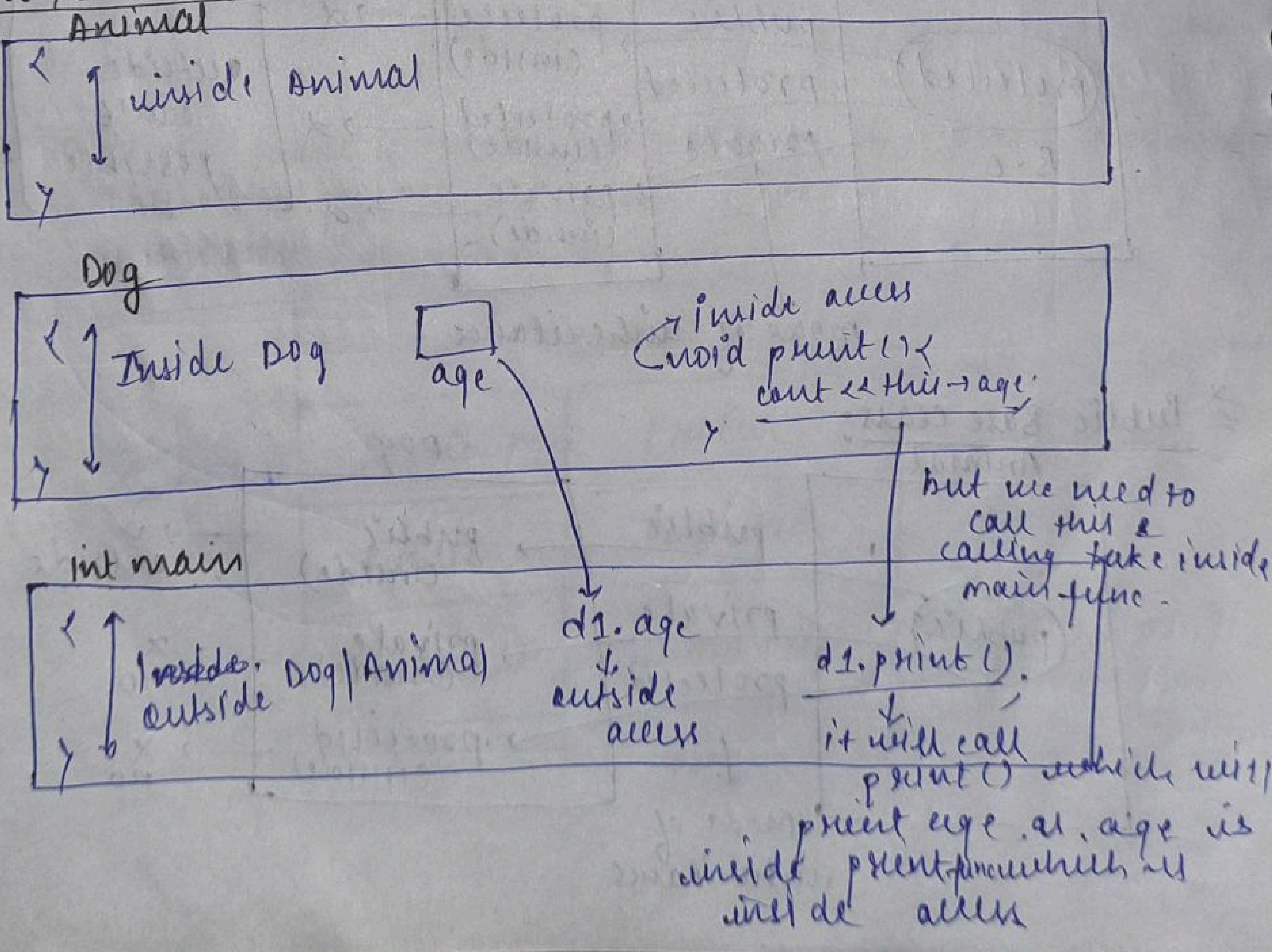




* Mode of Inheritance: It tells the ~~as~~scope of ~~d~~ member of derived class.

* 

Inside/outside Access

* 

but we need to call this & calling take inside main func.

it will call print() ~~which~~ will print age as age is inside print~~func~~which is inside access

* Type of Inheritance

① single ② Multi-level ③ Multiple ④ Hierarchial

⑤ Hybrid.

① Single-Level Inheritance:

when there is one Base class and one derived class.

i.e Parent class / Base class

↑

Base class / childclass / Derived class

ex: car (Parent class)

↑

scorpio (Derived class)

Syntax of single LI:

```
class scorpio : public car;
```

scorpio ↓ Derived class

car ↓ Base class

② Multi-Level Inheritance:

when one class inherit property from base class and then inherit class become base class for another class.

Involvement of more than 2 classes.

i.e Parent

↑

child

↑

Grandson

ex: Fruit

↑

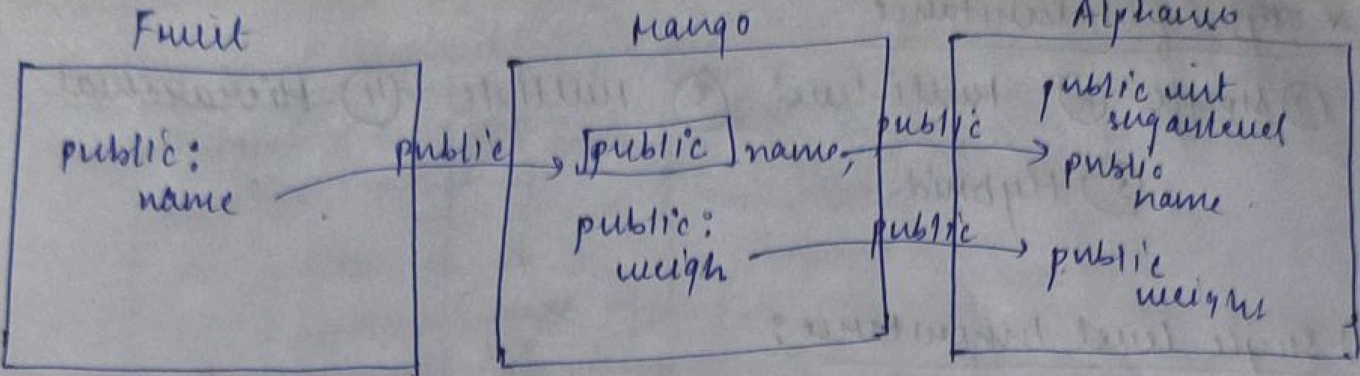
Mango

↑

Alphanso.

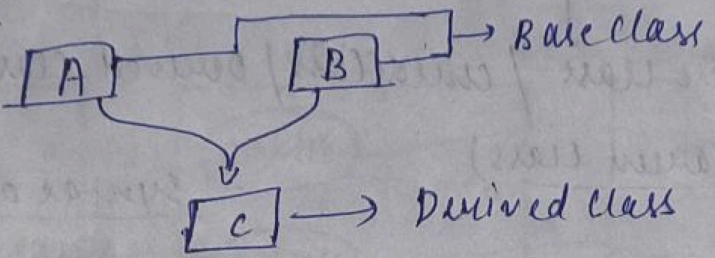
Syntax of Multi-Level Inheritance.

```
class A{

};
class B : public A{

};

class C : public B{

};
```

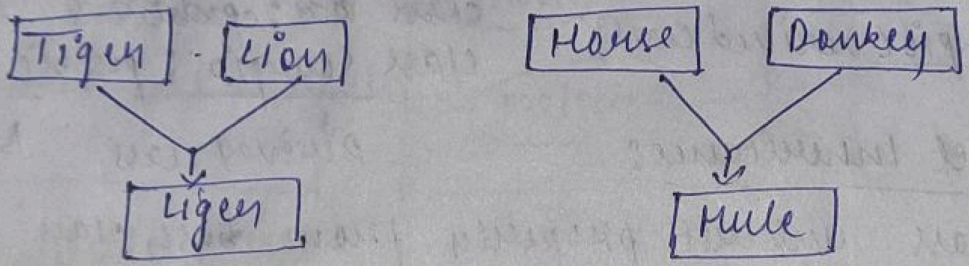


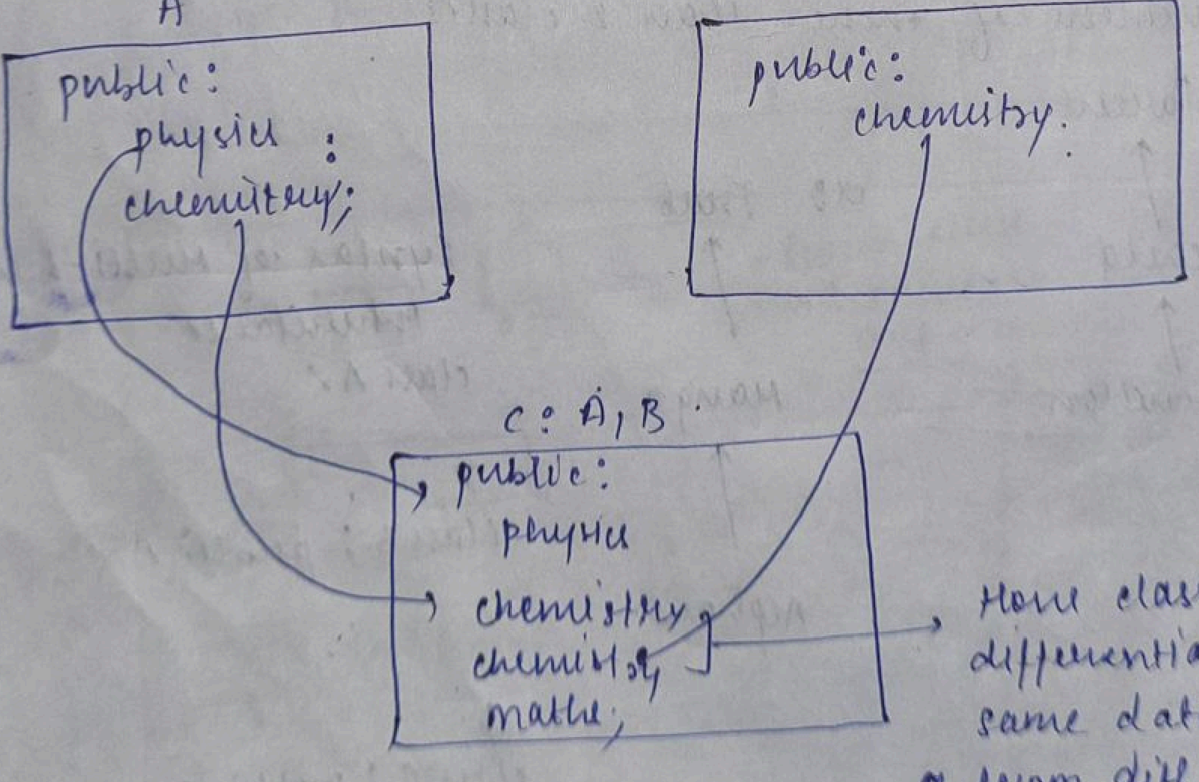
③ Multiple Inheritance:

A Derived / ~~Base~~ class which inherit properties and behaviour from two Base class said to be multiple Inheritance.



ex:





Hence class C will differentiate b/w same data member ~~g~~ from diff class.

* How class C will differentiate to call between ~~same~~ when same data members from 2 base classes different are inherited (Also known as Diamond Problem).

→ This is done with scope resolution operator ::

i.e ~~class~~ C obj;

```
cout << obj.A::chemistry
cout << obj.B:: chemistry
```

obj.A::chemistry → it will access the chemistry data member of class A.

obj.B::chemistry → it will access the chemistry data member of class B.

So with scope resolution operator we can access the the same data member of class A and class B which being inherited by class C.

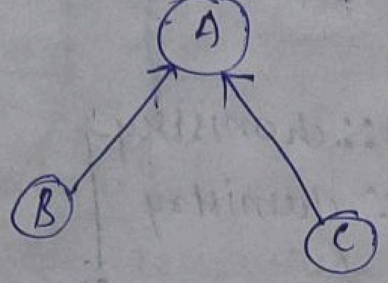
<u>Syntax of Multiple Inheritance</u>

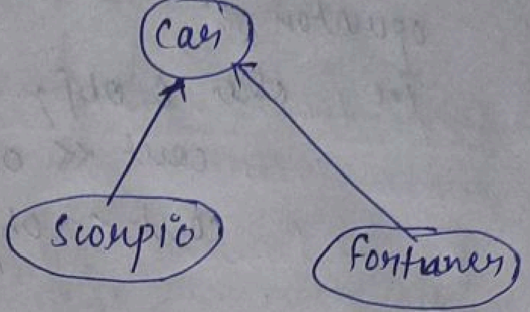
```
class A {

    // code

}

class B {

    // code

}
                public
class C : public A, B {

    // code

}
```

↳ two base class are separated by comma.

(4) Heirarchical Inheritance:

~~[illegible]~~ Two Derived class which inherit properties and behaviour from single Base class

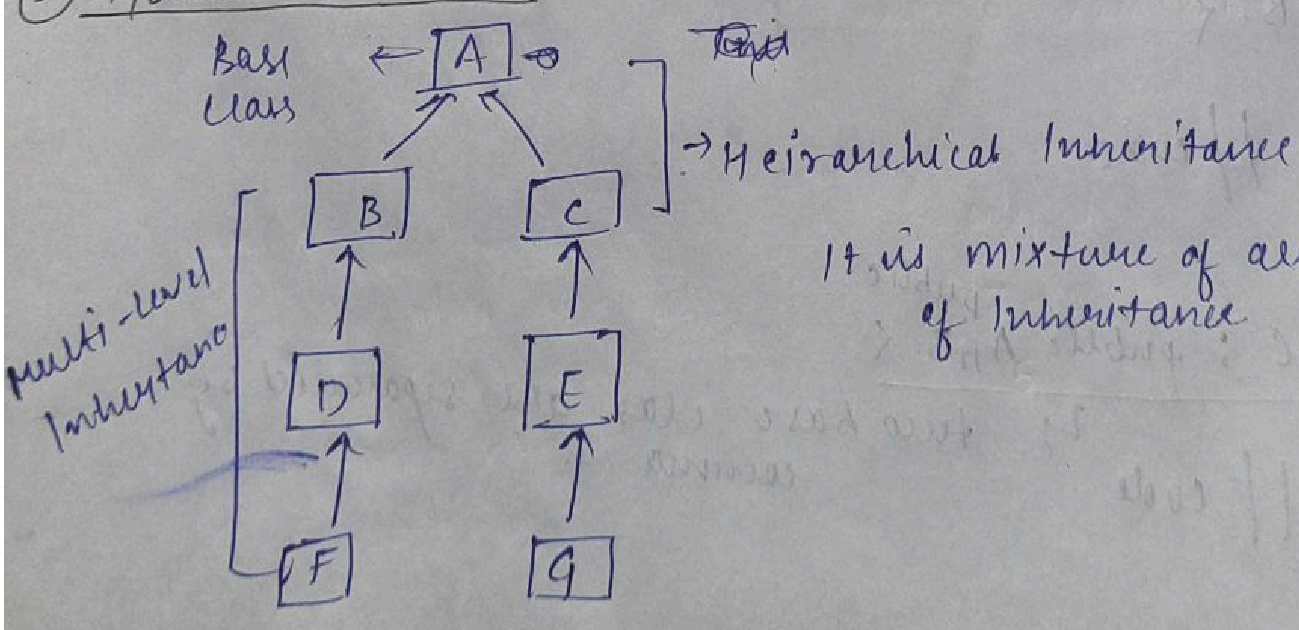




ex:

```
class A {

    //code.

}

class B : public A {

    // code

}

class C : public A {

    // code.

}
```

(5) Hybrid Inheritance:



It is mixture of all types of Inheritance

* Polymorphism:

Poly → many   existing in many forms
morph → form.

when a single func/entity with same name can perform different task i.e same thing existing in different forms.
func

* Type of Polymorphism:

① compile-Time Polymorphism.

② Run-Time Polymorphism

① compile-Time Polymorphism:

function Overloading   Operator Overloading

a) function Overloading:

function having the same name but different parameter said to be function overloading. Note that func must have same returntype.

ex:

```
int func( int a) {
    // code
}

int func ( int a,  int b) {
    // code
}

int func( int a,  string str) {
    // code
}
```

```
int main () {
    func(4).
    func(4, 10).
    func(4, "subrat").
}
```

b) operator overloading : ~~In that same operator~~ when an operator is used to operate different multiple type of operation.

ex: '+' operator is used for addition we can use it for subtractor. this is said to operator overloading.

syntax: returntype operator+ ( ) {

};

In operator overloding $\underset{\substack{\downarrow \\ \text{here a is} \\ \text{current} \\ \text{objec}}}{a}+\boxed{b.}$ → b is input value.

∴ a+b is equivalent to

$\underset{\substack{\downarrow \\ \text{object}}}{a}.\text{add}(\underset{\substack{\downarrow \\ \text{input valu}}}{b}) = a+b;$

code:
```
class Operator { ~~Coperator~~
    public:
        int val;
        void operator+(Operator &obj2){
                int value 1 = this → val;
                int value 2 = obj2. val;
                cout << (value 1 - value2) <<endl;
        }
};
int main(){
    Operator obj1, obj2;
      obj1.val=7.
      obj2.val=2;
      obj1+obj2;
```

output
```
7+2
5
```