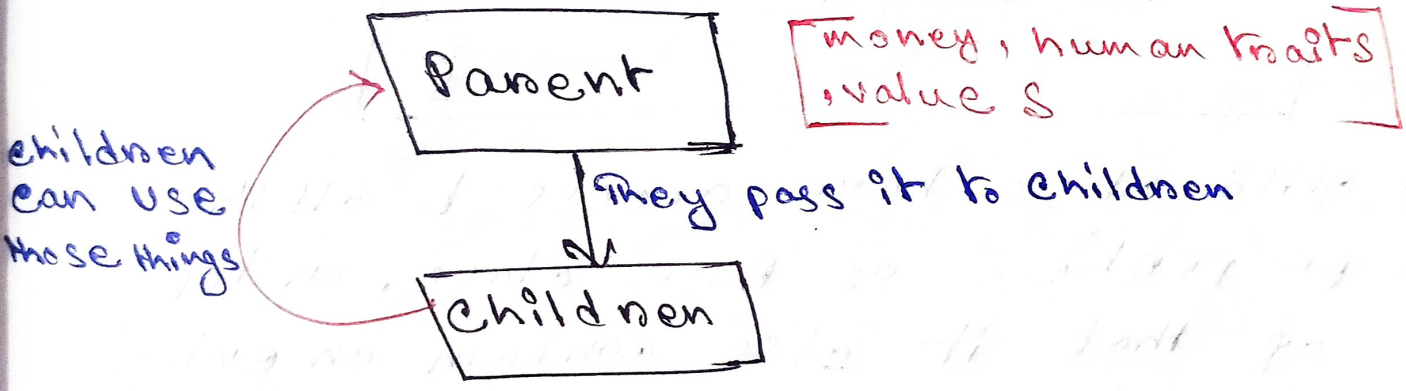
Object-Oriented-Programming #LEC-9

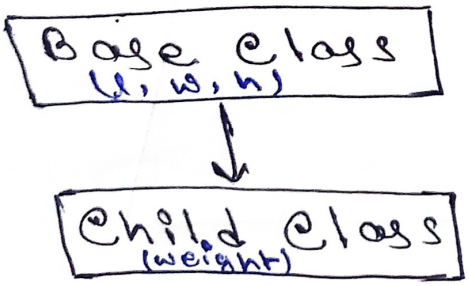
Properties of Object-Oriented-Programming:

Inheritance:

we can understand the concept of inheritance from real life. like, You have your parents and they own some properties on money on something and as a child of them you can inherited that from them



Now, in Object Oriented Programming sense, we can say if there is an class, which can inherit the properties and function of any other class



The child class inheriting the properties of base class, on top of that child class can have extra Arguments in it

so, now, how do child class actually inherit the properties of Base class
or
how do incorporate the definition of Base class into child class.

- 'extend' keyword : using extend keyword we can incorporate the definition of Base class into child class

```
class child extends Base{
    int weight
                        }
```

Now, child class have access to all the properties of Base class, on top of that it also contain an extra argument.

And, now we can access the properties of ~~child~~ Base in child class like this ⇩

```
child child = new child();
    child.l
    child.w
    child.h
```

and it will work

Example: let's say we have Species class

Now, the human child class can have the properties of base class, and also some additional properties of it's own

important: when you call this ⬇

```
child.l
```

it is being checked if the child class contain length or not

as, it does not, it will gonna checked it's parent Base class, as it Contain the length property, so it can return the length value and we can access it.

Now, whenever we do this ⇩

```
Human Bijoy = new Human();
Bijoy.age
```

then, the Human class trying access age from Species class, as it donot contains But inherited it from Species class

But, if we trying access a variable that is being present in the Parent class of a child class, that variable also need to be initialized.

```
child child = new child();
```

whenever we call a construction like this, we have to initialized parent class variable too.

- let's understand the concept of interitance through an example

you have created a Java file with name Box under which you have this class and it's construction.

```
public class Box {
double h;
double w;
double l;
public Box(){ //one of construction of
                class Box
this.h = -1;
this.w = -1;
this.l = -1;}

Box (double side){
this.h = side;
this.l = side;
```

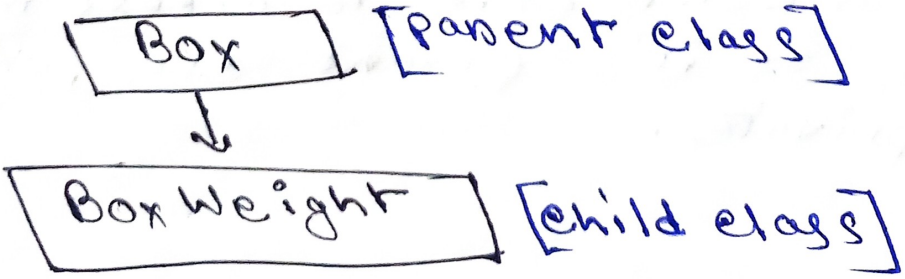
```
        this.w = side;
    }

    Box (Box old){
        this.h = old.h;
        this.l = old.l;
        this.w = old.w;}
}
```

under Main.java file

```
public class Main{
public static void main(String[] args){
    Box box1 = new Box(5); // it will call the
    constructor Box that takes no argument
// if we pass nothing as argument it will
    call that Box constructor, which not
    taking any parameter as argument init
    Box box2 = new Box(box1);
// it will call that constructor Box, which
    which takes another box as argument
```

- using 'extend' keyword:



So, previously we have created a 'Box' class, now we want to create a 'BoxWeight' name class which contain all the properties of Box class + extra argument in it. like this ⇩

```
public class BoxWeight extends Box {
    double weight; // it's own argument

    public BoxWeight() { // one of it's constructor, which pass no argument
        this.weight = -1;
    }

    public BoxWeight(double l, double h,
    double w, double weight) {
        super(l, h, w); // it's calling the parent class construction
    // used to initialise values present in parent class
        this.weight = weight;
    }
```

Note: class doesn't able to access the values of it's parent class, which is private.

```
public class Box {
    private double l;
    double w;
}
```

```
public class BoxWeight extends Box {
    double weight;

    public BoxWeight() {
        this.weight = -1;
        this.w;   ✓  // this will work
        this.l;   X  // this will not work as
                     it is being private in
                     ~~that~~ Parent class
```

Anything, that is private, you can only use it on that file, you can not use it outside that file.

- Subclass, Superclass:
  when reference to a subclass object assign to a super class variable, you only have access to those part of the object that is define in the parent class or super class

```
Box box5 = new BoxWeight(1:2, h:3, w:4,
                         weight: 8);
System.out.println(box5.w); // able to access it
System.out.println(box5.weight); // all no problem at
// can not access it
```

It is actually the type of the reference variable, and not the type of the object that determines what members can be access

reference type of Box, ~~***~~
referencing to BoxWeight

So, when reference to a subclass object assign to a superclass variable, you only going to have access only those part of the objects; that are obviously defined in the superclass

Now, what if we Swap the Box & BoxWeight

```
BoxWeight box5 = new Box(2, 3, 4);
```
// it will gonna give error

As, BoxWeight is reference variable So, we have access to weight parameters but as object type is Box which is a parent class in which there is no construction exist

which contain an initialize the Box parameter, So that's why error being occur.

Note:

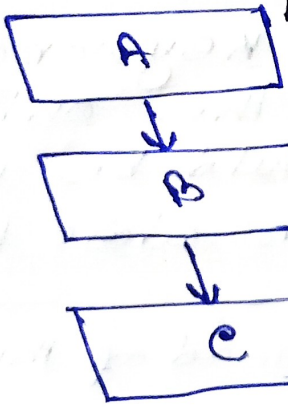
You can not have a child reference variable and parent object.

'super' keyword:

Whenever a subclass needs to refer to the super class or parent class from which it is defined, for that we use the super keyword.

- 'super keyword' also work on multi level classes

Example: if we have multi level inheritance, like, one class is inheriting other class, and so on



'super' keyword used in A class will be referring to object class

'super' keyword used in B class will be referring to A

'super' keyword used in C class will be referring to B

Note:

Class object is the root of the class hierarchy, Every class has object as a superclass

```
public object()
```

↑

This is a object() class presented by 'Java'

<u>usecase of the 'super' keyword:</u>

(1) Every single class inherits the object class, Even the BoxWeight child class also inherit the, but super keyword referring to the class above it.

here, Box class is directly above the BoxWeight class, and this is the first usecase of super keyword

(2) you can use super keyword as this keyword, but the only difference will be, it would be use to access the super class keywords like:

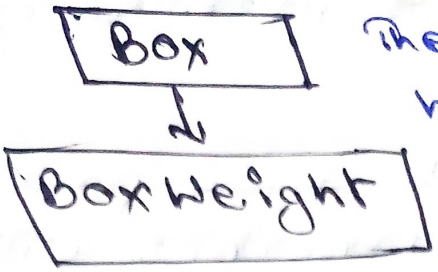
```
super.h   instead of this.h
```

? But why to use super instead of this.

→ Because, suppose both ~~class~~ parent and subclass has a variable with a same name, Suppose here Box class and BoxWeight class has a variable name weight then, both if you want mention or ~~want~~ specifically want to referring to the parent class weight variable in the child class, we ~~then~~ use 'super' keyword and instead 'this'

Code :-

```
public class BoxWeight extends Box {
    double weight;
    public BoxWeight() {
        this.weight = -1;
    }
    sout(super.weight); // here weight variable referring to the weight variable that is present in parent class
    sout(this.weight); // referring to the weight variable present in the child class itself.
```



The above class will not have the knowledge of the below classes

- in the BoxWeight child class, if we do this ↓

```
this.weight = weight;
super(h,w,d);
```

// this will not work

As, parent class doesn't care about what child class contains, but child class do care about what parent class contains, so we have to initialize parent class variables first like this. ↓

```
super(h,w,d);
this.weight = weight;
```

- Now, if do something like this, create a new construction in the child BoxWeight class

```
BoxWeight(BoxWeight other){
  super(other);
  weight = other.weight;  }
```

And, Previously we got this ↓ in the Parent class

```
Box (Box old) {
this.h = old.h ;
this.l = old.l ;
this.w = old.w ;
}
```

- here in the parent class construction taking Box type
- we are passing the BoxWeight type
- So, BoxWeight gonna have access to all the Box type values.

Internally: Box old = other. , of type BoxWeight

- even though the reference type is Box, BoxWeight gonna have access to Box types, which is exactly like this ↓

```
Box box5 = new BoxWeight(2,3,5,8) ;
```

these both are same

```
BoxWeight (BoxWeight other) {}
```

because, what is being accessed it depends on the type of the reference variable

but not in the ~~object~~ type of the object

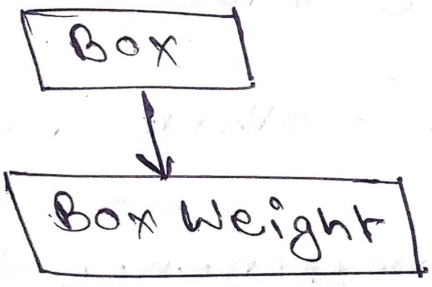
So, 'super' class variable can be used to reference any object from that derived class

hence, we are able to pass BoxWeight object to the Box construction

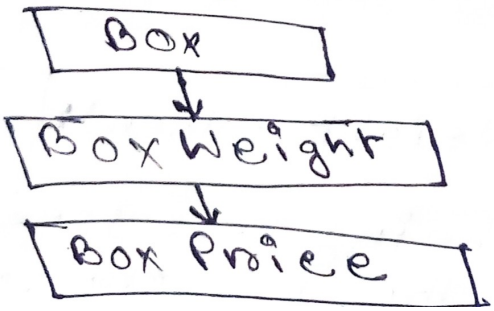
- Types of inheritance:

① Single inheritance: One class extends another class

ex:



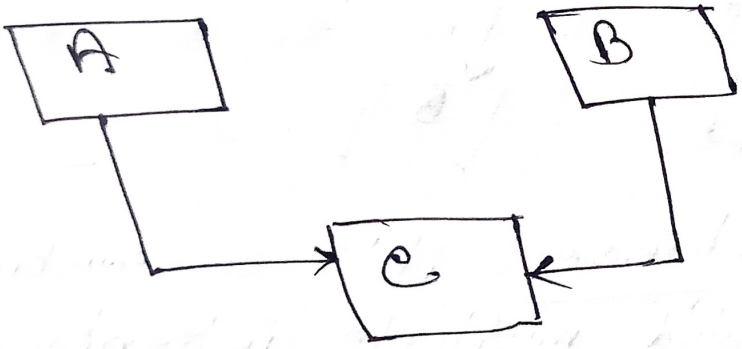
② Multilevel inheritance: In multilevel inheritance one class can inherit from a derived class, and then the derived class can become a parent for another new class



here, we can see BoxPrice is a Subclass of BoxWeight and BoxWeight is a Subclass of Box

Note:
Above classes has no idea of Bottom classes but Bottom class has idea about all the above classes

(3) Multiple inheritance: Incase of Multilevel inheritance one class inheriting more than one classes



here A and B both are parent classes so C has access to A and B both

Now, if A has n=5; and B has n=10;
and if we do something like this

C obj = new C();
c.n = ?

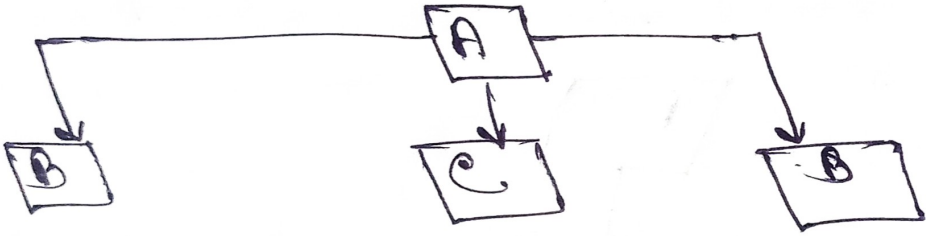
if two or more classes has same variable child variable doesn't know which one to pick
i.e Java doesn't support multiple inheritance

Note:

Multiple inheritance not allowed in Java

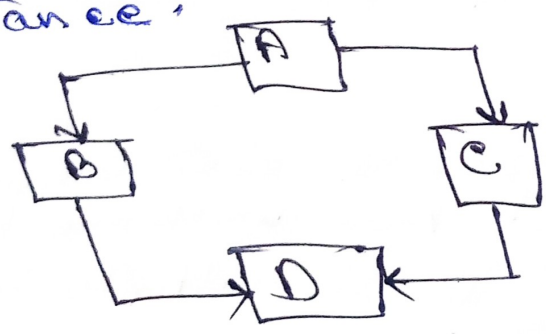
So, as multiple inheritance not supported in Java, we can still make a child class which have access to multiple parent class, using method called interfaces

(4) Hierarchical inheritance: One class is inherited by many classes.



(5) Hybrid inheritance: it's a combination of Single and multiple inheritance.

as, in Java we don't have concept of multiple inheritance, so we also don't have concept of Hybrid inheritance, as it consist the concept of multiple inheritance.



Polymorphism :-

Poly, Morphism

Poly → many

Morphism → ways to represent

So, Basically Polymorphism means many ways to represent a single entity or item.

Note :-
any language which doesn't support Polymorphism doesn't called object oriented language, instead they are called object Basis language.

let's say we have a parent class, shapes which has a method, like : area(). Now, shapes class is inherited by child classes like: circle, Square, Triangle.

- Types of Polymorphism :

① compile Time / static Polymorphism : it's Achieved via method called overloading

- method overloading : class has multiple method with the same name, but the number, type, order of the parameter and the return type can be different
Same name but types, argument, return types, ordering can be different

example of method overloading : multiple constructors.

```
A a = new A();
A a2 = new A(3,4);
```

- This is called compile time polymorphism because, Java decide which constructor or method will be called during compile time.
- if the name is same then the number of arguments, or the return type or the order should be different of the type

example:

```
int sum(int a, int b){
    return a+b; }

int sum(int a, int b, int c){
    return a+b; }
```

// Number of argument is different

or

```
void sum(int a, String b){
}

void sum(String b, int a){
}
```

// order of the type is different

Now, let's say we have something like ↓

```
double sum (double a, int b){
    return a + b;
}

obj.sum (2, 3);
```

// here we have double, we are taking arg as double

↳ // but here we are passing an integer

Note: when no matching is found Java automatically converts that integer into a double, this is called casting

(2.) Runtime Polymorphism / Dynamic Polymorphism

Achieved by method overriding.

- Overriding: when a child class has the method name same as the super class, method, parameter, and return type everything is same, just the body is different.

example:

```
public class Shapes {
    void area() {
        print("I am in square");
    }
}
```

```
public class Circle extends Shapes {
    void area() {
        print("Area is pie * r * r");
    }
}
```

// here we can see shapes is the parent

class and circle is the child class which is extending shapes, both has function called area, this is known as overriding.

Note:

if you want to check a method is overriden or not just put @Override above it.

- only the body ~~should~~ can be the same, ~~rest~~ ~~of~~

```
{
print("I am in Square") }
```

```
{
print("Area is Pi * r * r") }
```

Rest can be ~~different~~ Same

- How Overriding works:
  - If you have the type of the reference variable as the parent class, but the object is of type of the subclass
    - Overriding is done, when the reference variable you are using is of the superclass and which particular method will be called that depends on object.

```
parent obj = new child();
```

// here, which method will be called depends on this object

and this is known as Overriding

> What is how overriding works

- How Java determines which method to run :

Suppose, there is a function with the same name in the child class, and also there is function with the same name in the parent class, then how does Java determines which method to run?

- object type defines which one to run (So, by that we know child one will run)
- and the reference type will tell us which one to access.

But, how it happend, how Java determines this?

→ Java, Determines this by method Dynamic method dispatch

> Dynamic method dispatch: It is just a mechanism by which a call to an overriden method, that is resolved at run time rather than compile time.

★ • When the program is running during that time Java will determine, which method to run.

• We know a super class reference variable can refer to a subclass object. & when a overriden method is called through a super class sub variable, Java determines which version of that method to call, based on the type of the object

```
parent obj = new child();
```

obj: Superclass sub variable

child(): Type of the object

hence this determines at run time, when it's actually being called.

• So, it's type of the object not the type of the reference variable that ~~which~~ determines, which version of the overridden method will be executed.

■ "final" keyword: we know we use final keyword to create constants. but we can also use it to prevent overriding.

Suppose we have a method in parent class, and put final before the method

Parent class

```
public class Shapes{
  final void area(){          // this method can not be overriden
    print("I am in shapes");
  }
}
```

Sub class

```
public class Triangle extends Shapes{
  void area(){
    print("Area is 0.5 * h * b");
  }
}
```

// This will not work

as we have put final keyword before the method: void area(), it can not be overriden by other classes.

- Early Binding: The binding which can be resolve at compile time by the compiler is known as static or early binding.
  Binding of all the static private and final methods is done at compile-time.
- Late Binding: In the late binding or dynamic binding, the compiler doesn't decide the method to be called. Overri-

ding is a perfect example of dynamic binding.

> In overriding both parent and child classes have the same method.

- 'final' keyword can be also use to prevent inheritance

Parent class

```
public final class Box {
    ... code
}
```

Sub class

```
pubic class BoxWeight extends Box{
}
```

X

// this will not work, will give an error

- sometimes you want a class from being inherited for that (to prevent) you ~~want~~ use "final" keyword.

Note: when you put a class as final, implicitly it's methods became final too.

- ~~can we override static method~~

→ > static methods are not depend on the object of the class

> hence to access or edit static method, we don't have create

object out of it, we can do this directly using class name itself.

example:

Parent class

```
public class Box{
    .... some code
  static void greeting(){
   print("hey I am in: Box class, greeting!.");
                }
}
```

```
✗ box1.greeting();   // even though we can do it via reference variable, but we don't have to do this, as it is not conventional

✓ Box.greeting();    // we can do this directly using class name
```

- can we override static method.?

So, when we have another static method in other classes, so the idea is, can we override static method from Parent class to child class.

→ No, static methods are not or can not be overridden from parent to child class

All though static methods is being or can be inherited from Parent to child class, but can not be overridden as

As, there is no point of Overriding the static method, because it doesn't depend on the object.

- Overridding depends on object
- static methods does not depends on object.

hence, you can not override static methods

- In order to override something you have to deal with objects
- But static stuff eliminate all the objects.
- Dynamic method dispatch will not happend if it's static

- **Encapsulation**: Encapsulation means wrapping up the implementation of the data members and the methods inside a class
  - It basically hides the code and all the data, into a single entity or a unit so that can be protected from outside world; that is what the Encapsulation is all about.

▶ **Abstraction**: Abstraction means hiding the unnecessary details and showing the valuable information.

example:

Do you we really need to know all the mechanics of the car to run the car?

→ Obviously No; So all these information hidden from us, this is know as Abstraction.

another example:

when we, try to Print something, we just do this ⬇

```
System.out.println("Something you wanna print");
```

we really don't wanna know, what is actually happening on going on behind this, to print something using the print method.

- Abstraction vs Encapsulation:
  - Abstracting Solving a design level issue
  - Encapsulation Solving a implementation level issue.

> Encapsulation hides the code in the single entity and protected entire from outside world. Abstraction telling us ok, Encapsulation hidding things internally, I am Abstraction and I am providing you some of the

outside methods that you can use, in design way,
But how that thing is working internally that's handled by encapsulation

- Abstraction focuses on the external stuff
- Encapsulation focuses on the internal working.

> we can achieve Abstraction via
  — Abstract classes & interfaces

> In encapsulation we hide some details into a single entity, using internal
  — public, private, protected methods

- Abstraction is the process of gaining information.
- Encapsulation means process of containing the information.

- Data hiding vs Encapsulation:

- Data hiding focuses on data security
- Encapsulation focuses on hiding the complexity of the System.

example: hidded / Parent class

```
private double .l ;
```

// You will not able to access these data member outside the class file

```
private double l;
```

↑ This is actually data hiding

And, using the getters, setters is actually encapsulation

otherfile/child class

```
public double getL(){
    return l;          }
```

In Main

```
Box box = new BoxWeight();
box.getL();
```

- Data hidding: The concern is hiding the data security, along the complexity
- Encapsulation: ~~The con.~~ In Encapsulation the concern is wrapping the data to hide the complexity of the system

> in data hidding data should be private
> in Encapsulation data can be public, private etc.