Inheritance

To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword.

```
class subclass-name extends superclass-name {
    // body of class
}
```

You can simply specify one superclass for any subclass that you create.
Java does not support the inheritance of multiple superclass into one subclass.
You can, as stated, create a hierarchy of inheritance in with a subclass becomes a superclass to another subclass.
However, no class can be superclass of itself.

Although a subclass includes all the members of its superclass, it cannot access those members of the superclass that are declared as private.

A Superclass variable can Reference a Subclass object:

It is important to understand that it is the type of reference variable not the type of the object that it refers to that determines what members can be accessed.

When a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the Super class

```
plainBox = weightBox
(superClass)   (SubClass)

SUPERCLASS ref = new SUBCLASS();
// Here ref can only access method which are
available in SUPERCLASS.
```

Super keyword

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super.

Super has two general forms.
The first calls the superclass' constructor.
The second is used to access a member of the superclass that has been hidden by a member of a subclass.

```
Box(double w, double h, double d, double m)
{
  super(w, h, d); // call super constructor
  weight = m;
}
```

Box(double w, double h, double d, double m) calls super() with the arguments w, h and d. This causes the Box constructor to be called, which initializes width, height and depth using their value.

Box weight no longer initializes their values itself.
It only needs to initialize the value unique to it: weight. This leaves Box free to make these values private if desired.

Thus super() always refers to the superclass immediately above the calling class.
This is true even in a multileveled hierarchy.

```
class Box {
    private double width;
    private double height;
    private double depth;
    // construct clone of an object
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }
}
```

```
Class BoxWeight extends Box {
    double weight; // weight of Box
    // Construct clone of Object
    BoxWeight(BoxWeight ob) { // pass ob to
        super(ob);                Constructor
        weight = ob.weight;
    }
}
```

Super() is passed an object of type BoxWeight not of type Box.
This still invokes the constructor Box(Box ob).

Note - A superclass variable can be used to reference any object from that class.
Thus, we are able to pass a BoxWeight object to the Box constructor
Of course Box has only knowledge of its own variables

Second Use of super()

The second form of super acts somewhat like this except that it refers to the superclass of the subclass in which it is used.

super.member

Here member can be either a method or an instance variable. This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

super() always refers to the constructor in the closest superclass. The super() in BoxPrice calls the constructor in BoxWeight.
The super in BoxWeight() calls the constructor in Box.
In class hierarchy, if a superclass constructor requires parameters, then all subclass must pass those parameters "up the line".
This is true whether or not a subclass needs parameters of its own.

It makes sense that Constructors complete their execution in order of derivation.
Because a superclass has no knowledge of any subclass. any initialization it needs to perform is different from / separate from and possibly prerequisite to to any initialization performed by the subclass. Therefore it must complete its execution first.

NOTE: If, super() is not used in subclass Constructor, then the default or parameterless Constructor of each superclass will be executed.

Using final with Inheritance

The key word final has three uses:

First: used to create the equivalent of a named Constant:

Using final to Prevent Overriding.
To disallow a method from being overridden. Specify final as a Modifier at the start of its declaration.

Methods declared as ~~final~~ ~~for~~ final cannot be overridden.
Methods declared as final sometimes provide a performance enhancement:
The compiler is free to inline calls to them because it knows that will not be overridden by a subclass. When a small final method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code by the calling method. thus eliminating the costly overhead associated with a method call. Inlining is an option only with final methods.

Normally Java resolves calls to methods dynamically, at run time.
This is called Late binding. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is ~~called~~ called early binding.

Using final to prevent Inheritance:

To do this, precede the class declaration with final.

NOTE: Declaring a class final implicitly declares all its methods as final too.

As you might except, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself & relies upon its subclasses to provide complete implementations.

NOTE: Although static methods can be inherited, there is no point in overriding them in child classes because the method in parent class will run always no matter from which object you call it.

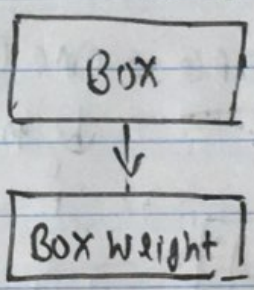
This is why static methods in interface cannot override them, they will always run the method in parent interface.

This is why static ~~method~~ interface method must have a body.

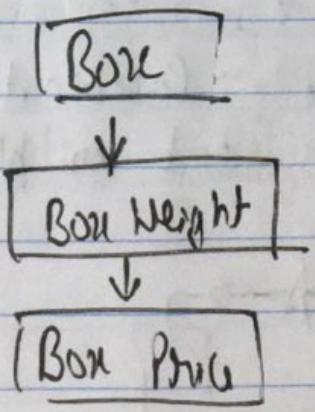
Note: Polymorphism does not apply to instance variables.

Types of Inheritance

1. Single Inheritance : one class extends another class

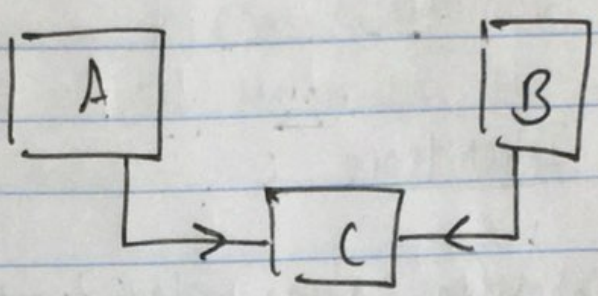


2. Multilevel Inheritance

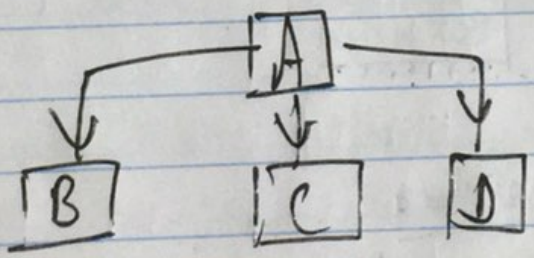


3. Multiple Inheritance :

One class extending more than one classes
(Not allowed in Java)

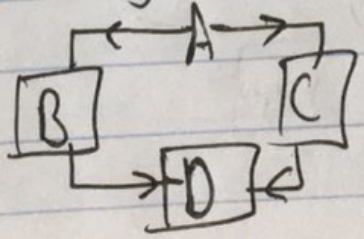


4. Hierarchial Inheritance : One class inherited by multiple classes.



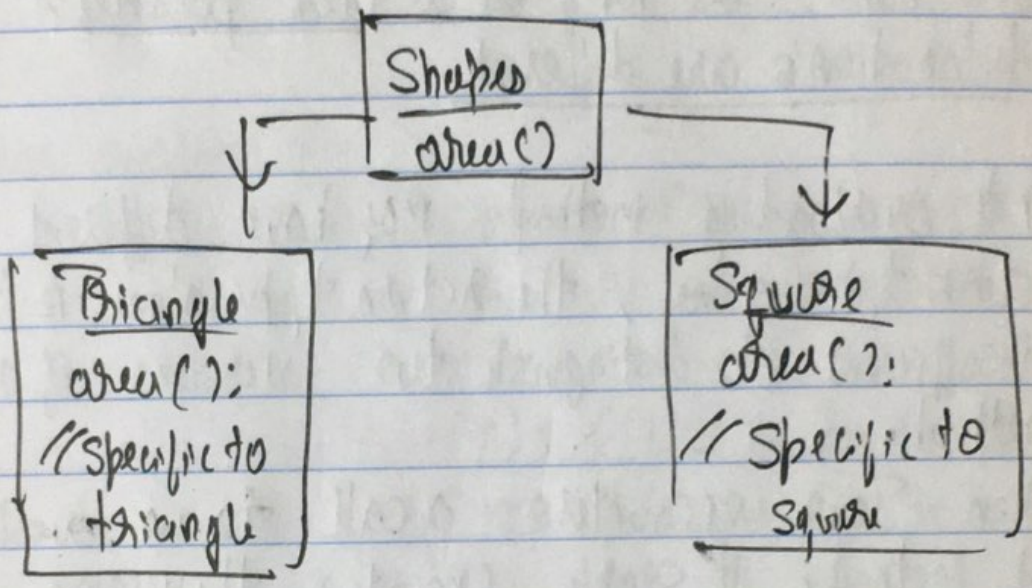
5. Hybrid Inheritance ; Combination of single and multiple inheritance.

(NOT in Java)



Polymorphism

many → Ways to ~~represent~~ represent



Types of Polymorphism:

① Compile Time / Static Polymorphism
⇒ attained via Method overloading

② Runtime / dynamic Polymorphism
⇒ attained via Method ~~or~~ overriding

Overloading Methods

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.

While overloading methods may have different return type alone, the return type alone is insufficient to distinguish two versions of a method.
When Java encounters a call to an overload-ed method, it simply executes the version of the method whose parameters match the arguments used in the call.

In some cases, Java's automic type conversion can play a role in overloaded resolution.

```
Class overload_demo {
    void test( double a){
        System.out.println ("Inside test (double)
                                a: " +a);
    }
}
Class overload {
    public static void main {String args[]}{
        overloaddemo ob. new overloadDemo()
        int i = 88
        ob. test(i); // will invoke test(doub
        ob. test (123.4)// will invoke test (doub
```

The ~~test~~ version of OverloadDemo does not define test(int).
Therefore when test() is called with an integer argument inside Overload, no matching method is found. However Java can automatically convert an integer into double, and this conversion can be used to resolve the call.
Therefore after test(int) is not found, Java elevates i to double ~~and~~ then calls test(double).

of course, if test(int) had been declared it would have been called instead.

Java will employ its automatic type conversion only if no exact match is found.

Returning the objects:

```
//Returning the objects.
  Class Test{
      int a;
       Test(int i){
          a=i;
       }
       Test incrObjTen(){
          Test temp=new Test(a+10);
          return temp;
       }
}
Class RetOb{
     psvm{
       Test ob1= new Test(2);
       Test ob2;
```

```
    ob2 = ob1.incrByTen();
    System.out.println("ob1.a: " + ob1.a);
    System.out.println("ob2.a: " + ob2.a);
  }
}
```

Output:

ob1.a: 2
ob2.a: 12

Each time incrByTen() is invoked, a new object is created, and a new reference to it is returned to the calling routine.
Since all objects are dynamically allocated using new, you don't need to worry about an object going out of scope because the method in which it was created terminates. The object will continue to exist as long as there is a reference to it somewhere in the program.
When there is no reference to it, the object will be reclaimed the next time garbage collection takes place.

Overriding

In ~~class~~ Class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in superclass.

When an overridden method is is called from within its ~~super~~ subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the super-class is hidden.

Method overriding occurs only when the names and the type signatures of the two methods are identical.
If they are not, then the two methods are ~~simply~~ simply overloaded.

Dynamic Method Dispatch

Dynamic Method Dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

Dynamic method dispatch is important because this is how java implements run-time Polymorphism.

Lets ~~start~~ begin by restating an important principle:
A Superclass reference variable can refer to a Subclass object.
When an overridden method is called through a superclass reference, Java determines which version of the method to execute based upon the type of the object being referred to at the time the call occurs.
Thus, this determination is made at run time.

In other words,
it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

If B extends A then you can override in A through B with changing the return type of method to B.

```
Parent obj = new Child();
```

Which method to call depend on ↩ Known as Upcasting.

Can we override static methods?

NO

If even if we call the 'obj' type Child(); of reference variable Parent, it will call the Parent Method. Not the Child() method.

Why?

"Static methods donot depend on objects"

⇒ "You can inherit, But you can't override
You can run, But you can't hide"

Encapsulation

Wrapping up the implementation of the data members & methods in a class.

Abstraction

Hidding the unnecessary details and Showing valuable information.

Abstraction	Encapsulation
Abstraction is the feature of oops that hides unnecessary details but shows the essential information.	Encapsulation is also a feature of oops. It hides the code and data into a single entity or unit so that the data can be protected from the outside world

It solves an issue at the design Level.	Encapsulation solves an issue at implementation Level.
It focuses on the external Lookout	It focuses ~~on teto~~ internal Looking
It can be implemented using abstract classes and interfaces	It can be implemented by using the access modifiers (private, public, protected)
It is the process of gaining information	It is the process of containing the information.
In abstraction we use abstract classes and interfaces to hide the code complexities	We use the getters and setters method to hide the data.