18/07/23

Principles of OOP — Inheritance, Polymorphism, Encapsulation, Abstraction

NOTE:

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

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

A Superclass variable can reference a Subclass Object.

It is important to understand that it is the type of the 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 superclass.

SUPERCLASS ref = new SUBCLASS(); Where ref can only access methods which are available in SUPERCLASS.

NOTE: You cannot do something like,
SUBCLASS ref = new SUPERCLASS();

→ Using super

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.

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

Here, BoxWeight() calls super() with the arguments w, h and d. This causes the Box constructor to be called, which initializes width, height and depth using these values. BoxWeight no longer initializes these 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 multilevel hierarchy.

(P.T.O)

29/7/23

```
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 an object.

    BoxWeight(BoxWeight ob) { // pass object to constructor
        super(ob);
        weight = ob.weight;
    }
}
```

Notice that 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 derived from that class.

Thus, we are able to pass a BoxWeight object to the Box constructor. Of course, Box only has knowledge of its own members.

29/7/23

→ Second Use of super

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

ex. instead of this.member, super.member
↳ helps when have variables with same name in superclass as well as subclass

Here, member can either be a method or an instance variable. This second form of super is most applicable to situations in which member names of a subclass hide member 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 a class hierarchy, if a superclass constructor requires parameters, then all subclasses must pass those parameters "up to the line." This is true whether or not a subclass needs parameters of its own.

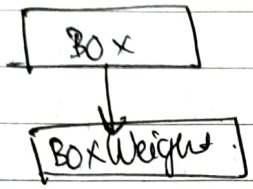
Even if we write super() in the first superclass, in this case Box() it still doesn't give error, because every superclass has Object() as its superclass.

If you think about it, 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 separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must complete its execute first.

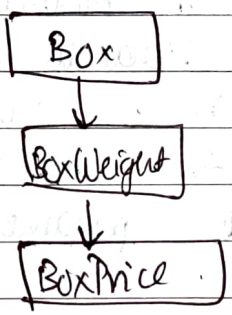
NOTE: If super() is not used in subclass constructor then the default or parameterless constructor of each superclass will be executed.

⇒ Types of Inheritance

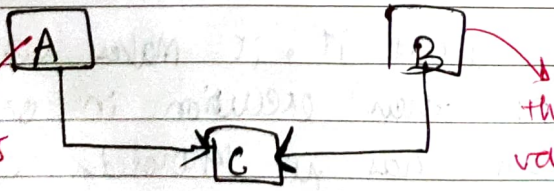
① Single Inheritance.
One class extends another class.



② Multilevel Inheritance
One class can inherit from derived class which will then become parent for another new class.



③ Multiple Inheritance {Not allowed in Java}.
When one class is extending more than one class.



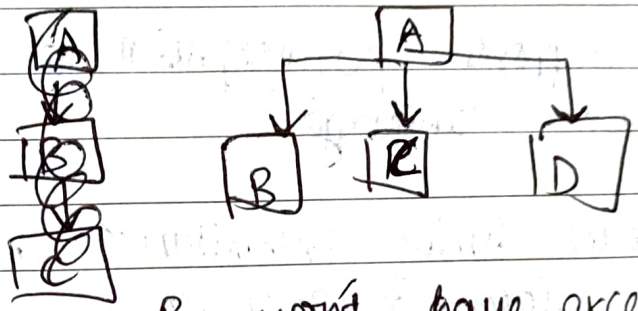
If you do,
C obj = new C();
C.n = ? → will it print 5 or 10 b/c both A & B are its parent class.

This is why Java does not support Multiple Inheritance. {Not allowed}

How will you do if multiple inheritance is not supported? You will use Interfaces which will be done later on.

④ Hierarchical Inheritance

One class is inherited by many classes.



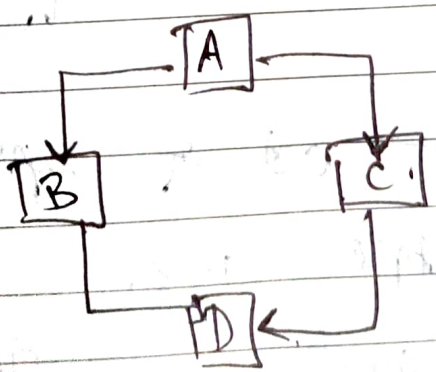
B would have access to C/D, similarly with C & D too. You can only access above.

Not go up, then below.

⑤ Hybrid Inheritance {we don't have this in Java}.

Combination of single and multiple inheritance.

{we will do it using interfaces}.



Polymorphism

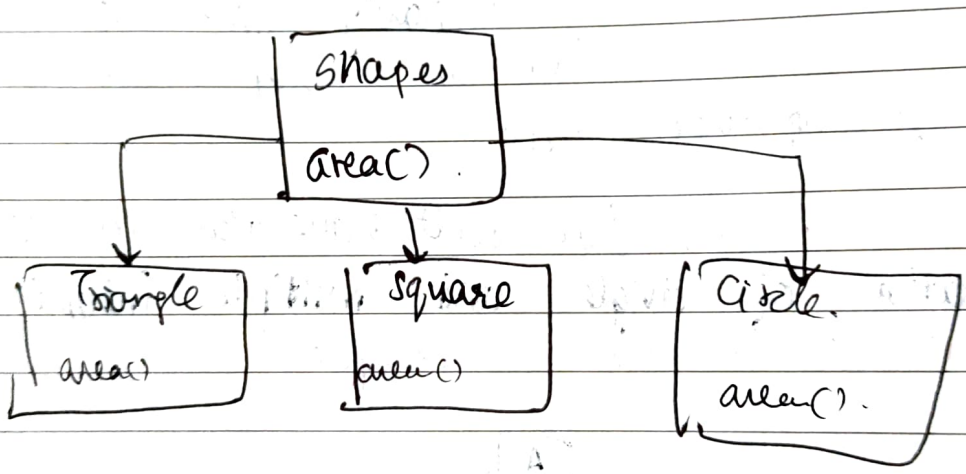
Poly (many) morphism (ways to represent)

In simple terms: So basically, polymorphism means many ways to represent single entity or item.

→ Java supports polymorphism, so it is object oriented language.

→ It occurs during inheritance b/c there are many classes related to eachother.

Ex.



In OOPS/Java **Polymorphism** → act of representing same things in multiple ways.

Means area() func^n is in every class, you can still access area() in different ways.

(check polymorphism in IntelliJ IDea)

```
Shapes shape = new Shapes();
shape.area();
```

Obviously it will give o/p according to the area() in shapes.

Say,
Shapes square = new Square ();
square.area ();
↳ it will give o/p from the area() in the square() class.

* why does it call square() class & not Shapes() ?

Types of Polymorphism

① Compile Time Polymorphism / Static Polymorphism
Achieved via method overloading

⇒ Method Overloading
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 overloaded methods may have different return types the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments in the call.

In some cases, Java's automatic type conversion can play role in overload resolution.

```
class OverloadDemo {
    void test(double a) {
        out("Inside test(double) a:" + a);
    }
}
class Overload {
```

```
psvm {
    OverloadDemo ob = new OverloadDemo();
    int i = 88;
    ob.test(i); //this will invoke test(double)
    ob.test(123.2); //this will invoke test(double)
}
}
```

As you can see, this 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 a 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 defined, it would have been called instead.
Java will employ its automatic type conversions only if no exact match is found.

Returning objects:

```
// Returning an object.

class Test{
    int a;
    Test(int i){
        a=i;
    }
    Test incrByTen(){
        Test temp = new Test(a+10);
        return temp;
    }
}
```

```
class Retobj
{
    psvm{
        Test ob1 = new Test(2);
        Test ob2;
        ob2 = ob1.incrByTen();
        sout(ob1.a);
        sout(ob2.a);
    }
}
```

o/p:

2
12

As you can see, each time incrByTen() is invoked, a new object is created, and a 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 your program. When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

② Run Time Polymorphism / Dynamic Polymorphism

Achieved by Method Overriding.

⇒ Method Overriding

In a 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 the superclass. When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by superclass will be hidden.

Method overriding occurs only when the ~~name~~ names and the type signatures of the two methods are identical. If they are not, then the two methods are 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. Let's 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 that 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 runtime.

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 a method in A through B with changing the return type of method to B.

Using 'final' with Inheritance:

- First, it can be 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 cannot be overridden.

Methods declared as final can sometimes provide a performance enhancement: The compiler is free to inline calls to them because it "knows" they 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 of 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 early binding.

- Using final to Prevent Inheritance.

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final.

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

As you might expect, 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 implementation.

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 interface methods cannot be overrided. Even if they override them, they will always run the method in parent interface. That is why static interface method must have a body.

Polymorphism does not apply to instance variables.

Encapsulation (Fundamental programming technique in OOP used to achieve data hiding)

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

Abstraction

Hiding unnecessary details and showing valuable information.

Abstraction

- Abstraction is a feature of OOPs that hides the unnecessary detail but shows the essential information
- It solves an issue at design level
- It focuses on the external lookout.
- It can be implemented using abstract classes and interface
- It is the process of gaining information.
- In abstraction, we can use abstract classes and interfaces to hide the code complexities.
- The objects are encapsulated that helps to perform abstraction

Encapsulation

- 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 implementation level.
- It focus on internal working
- It can be implemented by using the access modifiers (private, public, protected)
- It is the process of containing the information
- We use the getters and setter methods to hide the data
- The object need not to abstract that result in encapsulation.