Java OOPS

- OOPS (object - oriented Programming System)
  it is a methodology or paradigm to design a program using classes and objects. It aims to implement real world entities.

Class :- It is template or blueprint from which objects are created; it is a logical entity which defines common properties of objects belonging to that class.

Object :- It is basic unit of OOPS and represents real-life entity. It has state and behaviour which helps to represent any physical or logical entity.

4 pillars of OOPS

1) Abstraction
2) Encapsulation
3) Inheritance
4) Polymorphism

- Abstraction :- Hiding internal details and showing functionality is known as abstraction.
- Encapsulation :- Binding code and data together into single unit.
- Inheritance :- One object acquiring properties and behaviour of parent object.
- Polymorphism :- This feature allows object to behave differently in different condition.

Class, Objects & Constructors

Class :-

- Class is just a blueprint of object
- it doesn't occupy space

Object :-

- It is instance of class
- represent real world entity, also called instance

It consist of

a) State : reflects attribute of an object
eg for dog class, state are age, breed, color.

b) Behaviour : reflects methods of object
eg for dog, it is eat, bark etc.

c) Identity : It gives unique name to an object.
Eg for dog, it can be name of dog.

Notes

- Name of class should be PascalCase
- For abstraction prefer to make attributes private.
- To access private variables use getter and setter functions
- this keyword is used to denote entity of given class, this is only necessary when class variable and variable in any method has same name.

Code eg →

Car class → Car.java

Code

```
public class Car {
    private int doors;
    private String model;

    public void setModel (String model) {          // setter
     String validModel = model.lowerCase();
     this.model = validModel;
    }
    public String getModel() {                     // getter
        return this.model;
        // can even do return model
     }

}

// Initialize object
Car nexon = new Car();
nexon.setModel("nexon suv prime");
System.out.println(nexon.getModel());
```

Constructor :-

A constructor is a special method used to initialize objects in java. It runs first whenever object is created and initializes variables.

- it's name should be same as of class.

Constructor eg

Code

Q make class for complex numbers, make 2 constructors one with no parameters (default constructor) and one with two parameters (parameterized constructor)

```
public class ComplexNumber {

    private double real;
    private double imaginary;

    public ComplexNumber() {
        this(0.0, 0.0);
    }

    public ComplexNumber(double real, double imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

}
```

Note

1) Constructors don't have return type
2) One constructor can call other constructor using this
   we could have also done

```
this.real = 0.0;
this.imaginary = 0.0;
```

but this one is bad practice and code we write above in class is best way. We should prefer Constructor chaining it prevents code duplication.

Inheritance

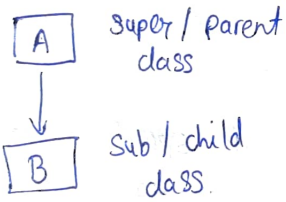
It is a mechanism in which one object acquires all the properties and behaviors of parent object.

It is IS-A relationship, also called parent-child relationship.

It helps in
- Method overriding (runtime polymorphism)
- Code reusability.

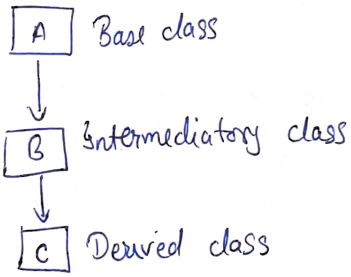
Types of Inheritance

a) Single inheritance

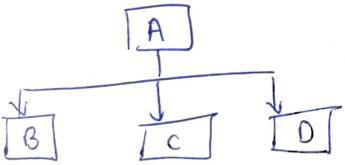



```
class B extends A {
}
```

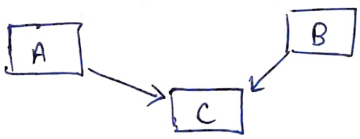
b) Multilevel inheritance



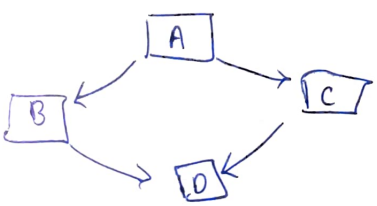
c) Hierarchical inheritance



4) Multiple Inheritance (through interfaces)



5) Hybrid Inheritance (through interfaces)



Code

Q make Dog class which extends Animal class and also override a method.

Base class

- Animal.java

```
public class Animal {
    private int size;
    private int legs;
    public Animal (int size, int legs) {
        this.size = size;
        this.legs = legs;
    }
    public void move (int speed) {
        System.out.println ("Animal move with speed " + speed);
    }
    public void eat () {
        System.out.println ("Its eating")
    }
}
```

- Dog.java

```
public class Dog extends Animal {
    private String hairs;
    private int weight;
```

```
public Dog (int size, String hairs, int weight) {
    super (size, 4);                    // initialize variables of super class
    this.hairs = hairs;
    this.weight = weight;
}

@Override                               // annotation
public void move (int speed) {
    System.out.println ("Dog moves with speed " + speed);
    super.move (speed);                 // to also call move of animal
    }
}
```

```
// Calling in main.java
Dog tommy = new Dog (120, "long golden", 80);
tommy.move (5);          // Dog move with speed 5
                            Animal move with speed 5
tommy.eat ();            // it's eating
```

- Keyword super is used to access / call parent class members (variables and methods)
- The call to super must be first statement in constructor.

Method Overriding vs method overloading (Polymorphism)

a) Overloading

- It means providing 2 or more seperate methods in a class with same name but different parameters.
  - > methods must have same name
  - > methods must have different parameters
- Methods can have different return types & modifiers.
- It is consider Compile time polymorphism, as compiler decide which method to call.
- Can overload static and instance methods.
- Usually happens inside a single class, but can also be treated overload in subclass if conditions are followed

b) Overriding

- It means defining a method in a child class that already exist in parent class with same signature (same name, same arguments).
- Also known as Runtime Polymorphism and Dynamic method Dispatch, because which method to call is decided at runtime by JVM.
- Can't override static methods.
- Constructors and private methods can't be overriden.
- Always happens between two classes

Static vs instance methods

a) Static methods

- declared using static modifier.
- They can't access instance variables and methods, so we can't use 'this' keyword in static method.
- They run by themselves without need to create object of that class
- Static methods are called as className.methodName(); or methodName(); only if in same class.

eg

Dog.java

```
public class Dog {
    - - -
    - - -
    - - -
    public static void printMe(){
    System.out.println("Hello");
    }
}
```

Main.java

```
public class Main {
public static void main(String[] args)
{
  Dog.printMe()
}
}
```

// so without creating instance of Dog class we can call it's static method.

* Instance methods can only be used for an instance of class.

- Static variable

Variable whose value will be same for all objects made from that class.

eg class A
static int val;

← object a
a.val = 3

- now object b is made its already val value is 3

Aggregation

If a class have an entity reference, it is known as Aggregation. It represents HAS-A relationship

eg:- Employee class contains many informations like id, name, email id etc. It contains one more object named Address, which is itself a class containing informations like city, state, country etc.

Code

```
class Employee {
    int id;
    String name;
    Address address;    // Address is itself a class
    ---
}
```

// Here Employee has an entity reference address, so relationship is Employee HAS-A address.

This is helpful in code reusability.

// let Address has method getCity(), and we want to use in object employee1.

so → employee1.getAddress().getCity();

final Keyword

final keyword is used to restrict the user.

- final keyword for variables

If you make variable as final, you can not change the value of it (it becomes constant)

eg
```
class Bike {
    final int speed = 90;
    void run(){
        speed = 100;
    }
}
```

```
Bike obj = new Bike();
obj.run();
```

> Output : Compile time error

- final method

If you make any method as final, you can not override it.

- final class

If you make class as final, you can not extend it.

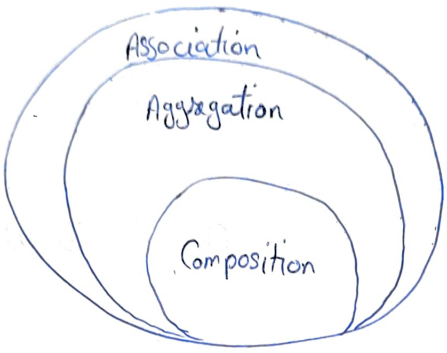
Notes

- final method can be inherited
  eg `final void run() { .. }`
- If final variable is uninitialized, it can only be initialized in constructor.

Association, Composition & Aggregation

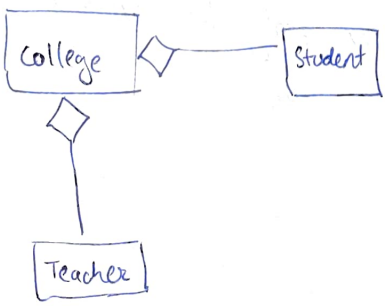
Association is a relation betn two seperate classes which establish through their objects. It can be one-one one-to-many, many-to-one, many-to-many.

Composition & Aggregation are two forms of association.



Aggregation

- HAS-A relation
- unidirection association
  One-way relation
  eg department has students but not vice versa
- Both entities can survive individually, which means ending one entity will not end other.



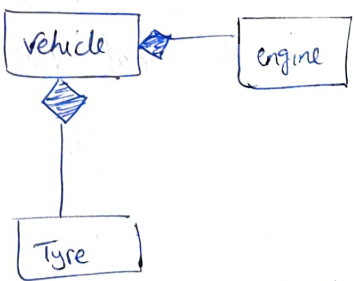
weak association

vs

Composition

- Part-of relationship
- Both entities are dependent on each other
- If there is composition betn two entities, compound object can not exist without other entity.

> Coding same as aggregation



vehicle can not exist without engine & tyre

- strong association

Encapsulation

Encapsulation allows programmer to hide and restrict access to data.

To achieve encapsulation :

1. Declare variables with private access modifier
2. Create public getters and setters that allow indirect access to those variables.

Why do we need encapsulation ?

1. Encapsulation allows us to modify the code or A part of code without having to change any other functions or code
2. Encapsulation controls how we access data
3. We can modify code based on requirements using encapsulation.
4. Encapsulation makes our application simpler

ABStraction

Hiding internal details and showing functionality is known as abstraction.

In java, we use abstract class & interfaces to implement abstraction.

Interface

- An interface in java is blueprint of a class.
- There can only be abstract methods & variables in interface not method body.
- It is used to achieve abstraction and multiple inheritance.
- Java interface represents IS-A relationship.

eg Phone.java ← interface

```
public interface Phone {
    void call(int number);
    boolean isRinging();
}
```

DeskPhone.java ← class

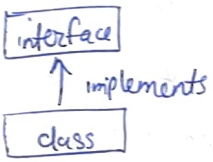
```
public class Deskphone implements Phone
{
    private int myNumber;
    private boolean isRinging;

    public DeskPhone(int number) {
        this.myNumber = number;
        this.isRinging = false;
    }
    @Override
    public void call (int number)
    {
        System.out.println("new call");
        isRinging = true;
    }
    @Override
    public boolean isRinging() {
        return this.isRinging;
    }
}
```

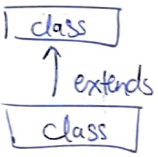
to use in main

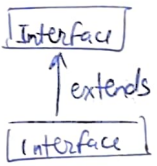
- Main.java

```
public class Main {
    public static void main (String[] args) {
        DeskPhone myPhone;
        myPhone = new DeskPhone(2345);
        myPhone.call(2345);
    }
}
```

// can even do Phone myPhone; without any other change.







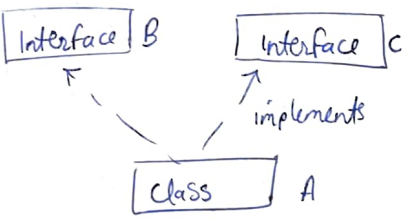
// All methods mentioned in Interface should be overridden in class.

> Multiple Inheritance can only be implemented via interfaces & not by classes

Reason

> Let interfaces B & C has same method run(); but since it is defined in A so no ambiguity.

But let B & C be classes with same methods run() and since A extends both so to use A.run(), it can't be able to select which method to use, either of B or C. So has ambiguity.



>> Public class A implements B, C {

}

Abstract class

- An abstract class is a class that is designed to be specifically used as base class.
- It can have abstract and non-abstract methods.
- It needs to be extended and its methods implemented.
- It can not be instantiated.

eg

Abstract class

Animal.java

```
public abstract class Animal
{
  private String name;

  public Animal (String name) {
      this.name = name;
  }

  public abstract void eat();
  public abstract void breath();

  public String getName() {
     return name;
  }
}
```

Dog.java

```
public class Dog extends Animal {
    public Dog (String name) {
        super (name);
    }
  @Override
   public void breath() {
       Sout ("Dog is breathing");
   }

  @Override
   public void eat() {
   Sout (getName() + " is eating");
   }
}
```

Can use abstract class methods without using super keyword.

So level of abstraction

1. Abstract class (0 to 100%)
2. Interface (100%)

abstract class does not support multiple inheritance

Inner Class (Nested class)

- Inner class is class defined inside the class or interface
- We use inner class to logically group classes & interfaces in one place to be more readable and maintainable.
- Additionally it can access all members of outer class, including private data & methods.

eg
```
class Java_outer_class {
    class Java_inner_class {
    }
}
```

- Inner class is part of nested class, non-static nested classes are known as inner class.

2 types of nested classes → static & non-static (inner classes)

Non static nested classes : 1) Member inner class
2) Anonymous inner class
3) Local inner class

a) Member Inner class : Class created within class & outside method

b) Anonymous Inner class : Class created for implementing an interface or extending class. Java compiler decides its name

c) Local Inner class : Class created within method

d) Static nested class : static class created within class

e) Nested Interface : Interface created within class or Interface

eg

> member inner class
(regular inner class)

```
class Outer {
    private int data = 30;
    class Inner {
        void msg() {
            System.out.println("data is " + data);
        }
    }
    public static void main(String[] args) {
        Outer obj = new Outer();
        Outer.Inner in = obj.new Inner();
        in.msg()
    }
}
```

> Anonymous inner class

Class that has no name. Should be used if you have to override method of class or interface

eg

```
abstract class Person {
    abstract void eat();
}
class Outer {
    public static void main(String[] args) {
        Person p = new Person() {
            void eat() { Sout("nice food"); }
        };
        p.eat();
    }
}
```

> Local inner class

Created inside a method & can only be invoked inside that method

eg

```
public class Outer {
    private int data = 30;
    void display() {
        class Local {
            void msg() { Sout (data);}
        }
        Local l = new Local();
        l.msg();
    }
```

> Static nested class

- It can access static data members of outer class, including private
- It can not access non-static (instance) data or methods.

eg

```
class Outer {
    static int data = 30;
    static class Inner {
        void msg(){ Sout(data); }
    }
    public static void main (String[] args) {
        Outer.Inner obj = new Outer.Inner();
        obj.msg();
    }
}
```