JAVA OOPs

INTRODUCTION TO OOPs

- It stands for Object - Oriented Programming System .
- It is a programming paradigm or methodology or technique where we create programs using classes & objects to solve a real - world problem .
- It maps and connects the code with the real - world .
- It also shortens the code and easy to understand .
- OOPs Languages : Java , C++ , Python , C# , etc .
- First OOPs language : Simula .
- First Purely OOPs language : Smalltalk .
- It increases reusability of code .

CLASS & OBJECT

▼ **Class**

- It is like a blueprint for creating objects .

- It is some sort of a template using which objects can be created .
- It is the collection of similiar objects.
- It is a logical entity .
- When we define classes , they won't take any space in memory ..

▼ **Object**

- It is a real - world entity like table , phone , rope , etc.
- It is the instance of a class .
- It is a physical entity .
- When we create objects , they take some space in memory .

```
Class : Vehicle          =>    Objects : Car , Bike , Bus , Trai

Class : Fruit            =>    Objects : Mango , Apple , Banana ,

Class : Car              =>    Objects : Scorpio , Volkswagen , I
```

OBJECT CREATION :

```
ClassName objName = new ClassName();       // Syntax to create
```

- Just like a real world entity has two things : Properties , and Behaviours .
- An Object has two things : Data Members (or Attributes) , and Methods .
- While a Class can contain : Attributes , Methods , Constructor , Nested Class , etc.

```
class Car {
    String brand ;           ---------
    double price ;                    |     Attributes or Data
    String color ;           ---------
```

```
    void accelerate(){       ----------
            // code                      |
    }                                    |
                                         |     Methods
    void stop(){                         |
            // code                      |
    }                        -----------
}

public class OOPs{
        public static void main(){

        // Object Creation
        Car c1 = new Car();

        // Accessing attributes
        c1.brand = "Ford";
        c1.price = 4500_000;
        c1.color = "white";


        // Accessing methods
        c1.accelerate();
        c1.stop();

        }
}
```

NEW KEYWORD

- It is used to create a new object in the heap memory .
- It returns reference to the newly created object .
- It allocates memory at runtime .

CONSTRUCTORS

- It is some blocks of code that is smiliar to methods .
- It is a special type of method , that is used to initialize object .
- It is called when instance of class is created .
- At the time of calling constuctor , memory for object is allocated .
- A constructor is called each time , as a new object is created .

TYPES OF CONSTRUCTORS :

▼ **Default Constructor**

- A constructor having no parameters .
- Also called no args constructor .
- It is used to provide default values to object .

▼ **Parameterized Constructor**

- A constructor having specific no. of parameters .
- It is used to provide different values to different objects .

```
class Employee{

        // some attributes

        Employee(){                         // Default Constructor
                // code
        }

        Employee(parameter1 , parameter 2 , ....){       // Para
                // code
        }
}

public class Constructor{
        public static void main(String[] args){
```

```
                Employee e1 = new Employee();        // construc
        }
}
```

> If you don’t create a constructor in the class , then compiler automatically creates default constructor .

RULES TO CREATE CONSTRUCTORS :

- Class Name and Constructor Name must be same .
- They don’t have any return type .
- They can’t be abstract , static , final , synchronized .
- But we can have access modifiers while defining constructor .

CONSTRUCTOR OVERLOADING

- Constructors can be overloaded just like methods .
- Multiple constructors with same name but different parameters , they are called overloaded constructors .

```
class Car{
    double price;
    String brand;
    String color;

    Car(){                  ------------------------------
        price = 50_00_00;                                  |
        brand = "Fortuner";                                |
        color = "white";                                   |
    }                                                      |
                                                           | Overl
    Car(double p , String b , String c){                   |
        price = p;                                         |
```

```
        brand = b;                                              |
        color = c;                                              |
    }                                                         |
                                                              |
    Car(String b){          -------------------------------
        price = 45_00_000;
        brand = b;
        color = "black";
    }
}

public class Constructors {
        public static void main(String[] args){

            Car c1 = new Car();
            Car c2 = new Car(32_00_000 , "Innova" , "Black" );
            Car c3 = new Car("Audi-A4");

        }
}
```

STATIC KEYWORD

- It is used for memory manangement .
- It can be applied to variables , methods , blocks , nested classes .
- It belongs to the class rather than instance of class .
- this and super cannot be used in static context .

▼ **Static Variable**

- Static variables also called Class variables .
- They get memory only once when class is loaded .
- It is used to refer the common property of all objects .

▼ **Static Method**

- It belongs to the class rather than the object of the class .
- A static method can be invoked without creating instance of class .
- A static method can access static property directly .
- A static method cannot access non-static property directly . It is done through creating instance of the class .

▼ **Static Block**

- It is used to initialize static data member .
- It is executed firstly as the first use of class is done and executed only once throughout the program .

```
class Employee{
    static String company;           // static variable
    int id;           // non-static variable

    static{                          // static block
            company = "Amazon";
    }
}

public class Static{
        public static void main(String[] args){

                Employee.company;          // accessing static v

                Employee obj = new Employee();
                obj.id;                   // accessing non-stati
        }
}
```

THIS KEYWORD

- A reference variable that refers current class object .

- It invokes current class constructor .
- It invokes current class method .
- It is passed as an argument in method call .
- It is passed as an argument in constructor call .
- It returns current class object .
- It refers current class instance variable .
- It cannot be used in static context .

```
class Human{
    int age;
    String name;

    Human(int age , String name){
        this();                  ---------         invoke constr

        this.age = age;          -----------|      refer current
        this.name = name;        -----------|
    }

    Human(){
        System.out.println("Default constructor");
    }


    void eat(){
            System.out.println("eats");
            this.walk();                 -------    invoke meth
    }

    void walk(){
            System.out.println("walks");
    }
}
```

```
public class Keyword{
        public class void main(String[] args){
                Human h1 = new Human(23 , "Yuvraj");
        }
}
```

PILLARS OF OOPS

- Abstraction ⇒ (simplying the user-interface)
- Encapsulation ⇒ (securing the data)
- Inheritance ⇒ (reusability of code)
- Polymorphism ⇒ (one entity but in multiple forms)

ENCAPSULATION

- A process where various components are put together into a single unit .
- Binding or Wrapping the data members into a single class .
- It restricts user from directly accessing the data members of a class .
- It can achieved by using acess modifiers and getter and setter methods .
- A way to achieve data binding . It secures our code .
- Example :

  ⇒ Just like various medicines are put together into a capsule .

  ⇒ Just like various departments are put together into an organization .
- JavaBean is an example of fully - encapsulated class .

IMPLEMENTATION :

- A class can be fully encapsulated by making data members private and using getter and setter methods to set and get the data .

- If we provide either setter or getter , then we can make our file read - only or write-only .

```
class Client{
    private long acNo;
    private String name;

    public void setDetails(long acNo , String name){         //
            this.acNo = acNo ;
            this.name = name;
    }

    public long getAcNo(){            --------
            return acNo;                        |
    }                                    |    getters
                                         |
    public String getName(){          --------
            return name;
    }
}


public class Encapsulation{
        public static void main(String[] args){
                Client c1 = new Client();
                c1.acNo;      // ----- error

                c1.setDetails(34652635537L , "Mohit");

                c1.getAcNo();
                c1.getName();
        }
}
```

INHERITANCE

- A process in which a class acquires or inherits properties of another class. It is called Inheritance.
- The class that gets inherited is Parent class or Super class or Base class .
- The class that inherits is Child class or Sub class or Derived class .
- Child class inherits attributes and methods from parent class .
- It represents IS-A relationship .
- Example : smartphone inherits properties and behaviours from normal phone .

ADVANTAGES :

- It increases reusability of code .
- It provides maintainability in code .
- Method Overriding or Runtime polymorpshism can be achieved .

IMPLEMENTATION :

- To implement inheritance in code , then we have to use " extends " keyword while declaring child class .

> 💡 In inheritance , when we create object of child class , then firstly constructor of parent class gets called , after then child class constructor gets called .

```
class Vehicle{
        String color;
        double price;

        void start(){
                System.out.println("starting....");
        }
    void accelerate(){
                System.out.println("accelerating...");
        }
```

```
}

class Car extends Vehicle{
        void reverse(){
                System.out.println("reversing...");
        }
}

public class Inheritance {
        public static void main(String[] args){

                Car c1 = new Car();
                c1.color = "black";
                c1.price = 15_000;

                c1.start();                     ------| Accessing
                c1.accelerate();                ------|
                c1.reverse();

        }
}
```

TYPES OF INHERITANCE :

1. Single Inheritance : (one parent class and one child class)
2. MultiLevel Inheritance : (multiple levels of inheritance)
3. Multiple Inheritance : (multiple parent class and one child class)
4. Hierarchical Inheritance : (one parent class and multiple child class)
5. Hybrid Inheritance : (combination of single and multiple inheritance)

In Java , mutiple inheritance and hybrid inheritance are not supported directly by class . They are supported through interface only .

SUPER KEYWORD

- It refers to the current parent class object .
- When you create instance of child class , then implicitly instance of parent class is created and refered by super keyword .
- It is used to refer current parent class instance variables .
- It is used to invoke current parent class methods .
- It is used to invoke current parent class constructor .
- It is used in methd overriding .

```
class Vehicle{

        public Vehicle(){
                System.out.println("vehicle constructor ....");
        }
        public void start(){
                System.out.println("starting ....");
        }

        public void stop(){
                System.out.println("stopping ....");
        }


}

class Car extends Vehicle{

        public Car(){
            super();                    -------  Invoking parent c
            System.out.println("car constructor ....");
        }
        public void functions(){
                super.start();          -------|   Invoking pare
```

```
                super.stop();           -------|
        }

}
public class OOPs{
        public static void main(String [] args){

                Car c1 = new Car();
                c1.functions();


        }
}
```

💡 In inheritance , while invoking constructor , super() call gets executed automatically , if no explicit constructor called . If it found default constructor in parent class , then no issues . But if not , then compile-time error occurs .

METHOD OVERRIDING

- If sub-class has same method as declared in super-class . It is called Method Overriding .
- It provides specific implementation to the method declared in parent class .
- It is used in run-time polymorphism .

RULES FOR METHOD OVERRIDING

1. Child class method name and parent class method name must be same .
2. Child class method parameters and parent class method parameters must be same .
3. Child class method return type and parent class method return type must be same .

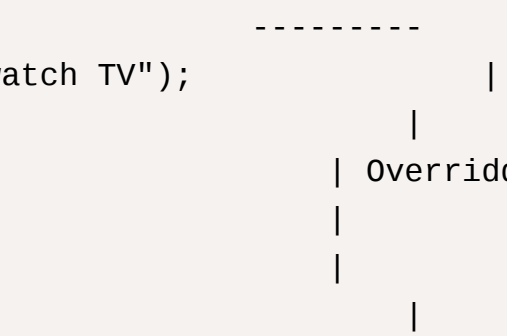
4. There must be inheritance .
5. While overriding , child class method must have less restrictive specifier than parent class method specifier .

> 💡 private > default > protected > public : Higher Restrictivity

```
class Parent{
        public void watchTV(){
                System.out.println("watch TV");
        }
}

class Child{
        @Override
        public void watchTV(){
                super.watchTV();
                System.out.println("watch cartoons");
        }
}
public class OOPs{
        public static void main(String[] args){

        }
}
```



METHOD OVERLOADING vs METHOD OVERRIDING

Method Overloading

- increases readibility .
- it occurs within the same class .
- compile time polymorphism .

Method Overriding

- provides specific implementation .
- occurs b/w 2 classes implementing inheritance .
- run-time polymorphism .

- inheritance not required .
- return type does not matter .
- method names must be same but different parameters .

- inheritance required .
- return type must be same or co-variant .
- method names , parameters , return type , all must be same .

FINAL KEYWORD

- It is used to restrict the user .
- It is used in multiple contexts : variable , method , class .
- It is a non-access modifier .

FINAL VARIABLE :

⇒ If you declare any variable final , you cannot change the value of that variable . It will be constant .

⇒ It is mandatory to initialize final variable . It cannot be left uninitialized .

⇒ If you can't initialize variable while declaring , then you must have to initialize it in constructor .

```
class Bike{
        private final double mileage = 21.3 ;             // fi
        mileage = 23.5;         // can't modify final variable

        private final String name;

        Bike(){
                name = "Splendor";   // can initialize final va
        }
}
public class OOPs{
        public static void main(String[] args){

                Bike b1 = new Bike();
```

```
        }
}
```

FINAL METHOD :

⇒ If you make any method final , then you cannot override that method .

```
class Bike{
        public final void run(){                    // final met
                System.out.println("Bike is running ....");
        }
}

class Honda extends Bike{
        @Override
        public final void run(){                    // error , c
                System.out.println("Honda Bike is running ....")
        }
}


public class OOPs{
        public static void main(String[] args){

                Honda h1 = new Honda();
                h1.run();               // it will execute and c
        }
}
```

FINAL CLASS :

⇒ If you make any class final , then you cannot extend or inherit that class .

```
final class Bike{                               // final class
        public final void run(){
                System.out.println("Bike is running ....");
```

```
        }
}

class Honda extends Bike{                       // cannot extend

}

public class OOPs{
        public static void main(String[] args){
        }
}
```

💡 Many pre-defined java classes are declared final : Wrapper classes , String class , Math class , System class , etc.

POLYMORPHISM

- It allows us to perform an action in different ways .
- An entity can have multiple forms , thus we can use that entity in different ways .
- Example : smartphone act as a camera , phone , speaker , etc.
- It simplifies our code .

TYPES OF POLYMORPHISM :

1. Compile - Time Polymorphism (Static Polymorphism)
2. Run - Time Polymorphism (Dynamic Polymorphism)

STATIC POLYMORPHISM :

- It is achieved through Method Overloading or Operator Overloading .
- But Java does not support Operator Overloading .

```
// Achieving Compile - Time Polymorphism

        public void add(){                          --------
            int n1=7 ;                                      |
            int n2=9;                                       |
            System.out.println(n1+n2);                      |
        }                                                   |  Over
                                                            |
        public void add(int n1 , int n2){                   |
            System.out.println(n1+n2);                       ---------
        }

        public int add(){                          ----- Not Overl
                // code
        }
```

BINDING :

- A mechanism using by which the compiler decides which method call will execute which method body . This mechanism is called Binding .
- Binding is connecting method call to method body .
- Types of binding :
  1. Static Binding (Early Binding)
  2. Dynamic Binding (Late Binding)

STATIC BINDING :

- When type of object is determined at compile - time . It is called static binding .
- When there is static , private or final method in a class , then there is static binding .
- Here , method call is decided by reference .

DYNAMIC BINDING :

- When type of object is determined at run - time . It is called dynamic binding .
- Here , method call is decided by object .

```
class A{

        public static void method(){
                // code
        }

        public void display(){
                // code
        }
}

class B extends A{

        public static void method(){            // Method Hiding
                // code
        }

        public void display(){                 // Method Overrid
                // code
        }

}
public class OOPs{
        public static void main(String[] args){

                A obj = new A();
                obj.method();             // parent class wala m
                obj.display();            // parent class wala d

                A obj = new B();
                obj.method();            // parent class wala me
```

```
                obj.display();          // child class wala dis

                B obj = new A();
        }
}
```

```
ParentClass obj = new ChildClass();     // valid
ChildClass obj = new ParentClass();     // invalid
```

RUN-TIME POLYMORPHISM :

- It is also called Dynamic Method Dispatch .
- To achieve it , we must be able to use same reference to call different versions of the same method .
- The reference will be of Parent class and the versions of the method will be defined or overridden in Child classes .
- It makes our code simple and efficient on the basis of memory consumption .

```
class Language{
    public void greetings(){

    }
}

class Hindi extends Language{
        @Override
    public void greetings(){
        System.out.println("Namaste");
    }
}

class English extends Language{
        @Override
```

```
    public void greetings(){
        System.out.println("Hello");
    }
}

class French extends Language{
        @Override
    public void greetings(){
        System.out.println("Bonjour");
    }
}


public class OOPs{
    public static void main(String[] args){

            Language obj = new Hindi();
            obj.greetings();

            obj = new English();
            obj.greetings();

            obj = new French();
            obj.greetings();
    }
}
```

ABSTRACT CLASS

- A class can be declared using abstract keyword . This type of class is called Abstract class .
- It can have abstract methods as well as non-abstract methods .
- It cannot have objects or It cannot be instantiated .
- It can have constructors , static methods , and final methods .

ABSTRACT METHOD

- A method which is declared abstract and having no implementation , is called abstract method .
- If a method is abstract , then its class must be abstract .

> 💡 When you are extending an abstract class having some abstract methods , then it is must to implement those abstract methods in this concrete class .
> If you don't give implementation to those abstract methods , then you must have to declare that class abstract .

SOME POINTS RELATED TO ABSTRACT :

1. abstract is a keyword .
2. It is used to declare the methods that don't have any implementation .
3. If method is declared abstract , then its class also must have to be abstract .
4. But you cannot abstract static methods , constructors , final methods , private methods .
5. abstract class don't have objects but have constructors .

```
abstract class Language{
    abstract public void greetings();
}

class Hindi extends Language{
        @Override
    public void greetings(){
        System.out.println("Namaste");
    }
}

class English extends Language{
```

```
        @Override
    public void greetings(){
        System.out.println("Hello");
    }
}

class French extends Language{
        @Override
    public void greetings(){
        System.out.println("Bonjour");
    }
}


public class OOPs{
    public static void main(String[] args){

            Language obj = new Hindi();
            obj.greetings();

            obj = new English();
            obj.greetings();

            obj = new French();
            obj.greetings();
    }
}
```

INTERFACE

- It is the blueprint of a class .
- It can contain data members and methods just like classes . But they do not contain constructors.

- Java automatically adds public static final before any data member while declaring it .
- Java automatically adds public abstract before any method while defining it .
- From Java 8 , interface can have “default” and “static” for non-abstract methods .
- From Java 9 , interface can have “private” for non-abstract methods .
- They contain abstract methods only but since Java 8 we can have non-abstract methods as well .

SOME POINTS :

1. It is used to achieve abstraction and multiple inheritance .
2. It also represent IS-A relationship .
3. It cannot be instantiated just like abstract classes . But its reference can be created .
4. When a class inherits an interface , then “implements” keyword is used there .
5. If a class inherits an interface , then it is must for class to provide implementation to all abstract methods defined in interface or either define that class abstract .
6. An interface cannot inherit any class . But it can inherit intefaces .
7. They are slower and limited . They are used to achieve loose coupling .

IMPLEMENTATION :

```
interface Language{
    void greetings();
}

class Hindi implements Language{
        @Override
    public void greetings(){
        System.out.println("Namaste");
```

```
    }
}

class English implements Language{
        @Override
    public void greetings(){
        System.out.println("Hello");
    }
}

class French implements Language{
        @Override
    public void greetings(){
        System.out.println("Bonjour");
    }
}


public class OOPs{
    public static void main(String[] args){

            Language obj = new Hindi();
            obj.greetings();

            obj = new English();
            obj.greetings();

            obj = new French();
            obj.greetings();
    }
}
```

ABSTRACT CLASS vs INTERFACE

Abstract Class	Interface

- It can contain abstract methods as well as non-abstract methods .
- It does not supports multiple inheritance .
- It can have static , non-static , final , non-final data members .
- They can provide implementation of interfaces .
- abstract keyword is used .
- They can inherit a class and multiple interfaces .
- By using them , we can achieve 0-100% abstraction .

- It contain only abstract methods . Since Java 8 , it can have non-abstract methods using default or static .
- It supports multiple inheritance .
- It can have public , static , final data members .
- They cannot provide implementation of classes .
- interface keyword used .
- They can only inherit interfaces .
- By using them , we can achieve 100% abstraction .

MULTIPLE INHERITANCE USING INTERFACE :

- In Java , multiple inheritance is not supported through classes , but supported through interfaces .
- When an interface extends multiple interfaces or when a class implements multiple interfaces , this is known as multiple inheritance .
- When class inherits another class , then we use “extends” keyword .
- When interface inherits another interface , then we use “extends” keyword .
- When class inherits interface , then we use “implements’ keyword .

ABSTRACTION

- It is a process of hiding implementation and only showing functionality to users .
- Here it deals with what the object do , not how the object do .

- Example : we use remote , we only get functionalities through keys , not any internal details .
- It provides simple user interface .
- Hiding internal details and only showing essential information and functionalities to users .
- It can be achieved through abstract class and interface .

ACCESS MODIFIERS

- They are keywords in Java .
- They specify visibility or accessibility of classes , methods , variables , etc. within a program .
- They control how these elements can be accessed from other parts of the program .

TYPES OF ACCESS MODIFIERS :

1. public
2. private
3. default (it is not a keyword , if you not mention any access specifier explicitly , then compiler use this)
4. protected

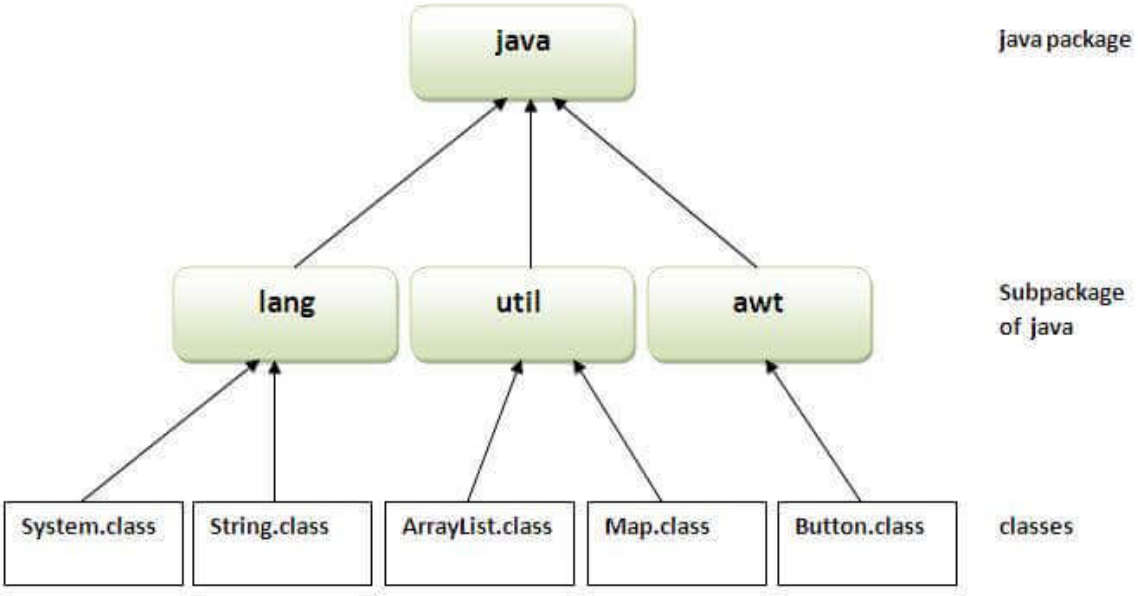
Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

PACKAGES

- They are group of similiar classes , interfaces and sub-packages .
- Two types of packages : Built in packages and user-defined packages .
- Built-in packages are java , lang , util , awt , javax , swing , etc.

ADVANTAGES :

1. organize the classes and interfaces .
2. provides access protection .
3. removing naming collisions .



ACCESS PACKAGE FROM ANOTHER PACKAGE :

- import package.*; ⇒ (all the classes and interfaces of package gets accessed but not sub-packages)
- import package.classname; ⇒ (only declared class of package accessible)

- using fully qualified name ⇒ (only declared class of package accessible , no need to import)

```
domain.package.subpackage.class ;                    // Syntax to name packages
```