# Object Oriented Programming in Java

10/9/24

## Method :-

A Function written inside a class with Oops Concepts.

Method = Function + Oops

↳ In Java every function Comes only inside the Class, hence they are only known as Methods.

↳ Hence there are no functions in Java as everything is will be called as "Methods".

| Function definition [Behaviour / logic] | vs | Function Calling [Argument] |
|---|---|---|
| A Block of Code that describes what the function does | | A "function call" is when you execute (or invoke) the function that was defined. |
| Can be Called multiple times | | when called, the control goes to the function definition. |

Eg :-

```
Public class FunctionExample {

  // Function Definition

  Public static int addNumbers (int a, int b) {

      return a+b;

  }

  Public static void main (String[] args) {
    // Function Calling
    int Sum = addNumbers (5,10);

    System.out.println(Sum);

  }

}
```

Void → does not return any value
→ Performs the task inside the function.
→ Prints anything inside the function if SOP is Present.

Not Void (OPP)

Parameterized function ⇒ Parameters / Arguments are given inside the function Call.

Non-Parameterized function ⇒ No Parameters / Arguments are given inside the function Call,

Parameters / Arguments will be present inside the function definition.

# Recursion

A Function Calling itself inside a function definition is called as "recursion".

Base Case :- The time when the Program will terminate

Recursion Case :- The time when the Program Calls itself.

Applications :-

i) Tree Traversal (BST) [In-order Traversal]

ii) Divide and Conquer Algorithms [Merge sort]

iii] Graph Algorithms (DFS)

iv] Gaming [Chess or tic-tac-toe]

=> Reduces the need for Complex loops and Conditions.

=> Can lead to high memory usage due to stack depth (stack overflow)

# Class

Eg:- Application Form

Blank Application Form => Class

Specific ,, ,, => object ← Instance (copy)

Class:-

→ Blueprint or template for Creating objects

→ Consists of Fields (Attributes/Variables), Methods (Functions), Objects. [Members of a class]

Eg:-

```
AppForm Codeio = new AppForm();
```

↓

here new is given for memory allocation purpose for the Created object.

## Access Modifiers

→ Keywords used to set the visibility or accessibility of classes, methods, Constructors and Variables.

→ They define how the members of a class can be accessed from other classes or objects.

Why does inside class Main Method there will be always "Public" static and not any others keyword Access Modifiers?

The Reason is the Compiler which runs outside the Package should always have the access of that Particular Code.

Setter Function:-

If an information appears to be private but we want it to access, then we create a function which can access that Private information of another class, then this function is known as "Setter Function".

Getter Function:-

Used to get the Values from another Class which may / maynot be Public

Eg:-

Both the Function's are more Important in Banking System's, data management of Patients etc,....

| Modifier | Class | Package | Subclass | World |
|---|---|---|---|---|
| Private | Yes | NO | NO | NO |
| default | Yes | Yes | NO | NO |
| Protected | Yes | Yes | Yes | NO |
| Public | Yes | Yes | Yes | Yes |

Private: Most restrictive, only accessible with the same class.

default: Accessible only within the same Package

Protected: Accessible by same package & by subclasses in other Packages.

Public: less restrictive, accessible from anywhere

Real-World Analogy :-

Public => Eg:- Public Park

Private => Personal bedroom

default => Anyone inside the house can access it, but outsiders can't

Protected => your family can access it even when they are in another house. eg:- myself

# Constructors

object Creation Syntax:-

Initialize Method

Classname object name => new Classname();

new → Memory allocation

Classname() → Constructor

=> Basically has the same name as the class

↳ A Method to initialize objects, it is called when an object of a class is created.

↳ No Return type [Not even void]

↳ Automatically called when an object is created.

Types of Constructors:-

(i) Default Constructor:-

↳ It is the one that takes no arguments

(ii) Parameterized Constructors:-

↳ A Constructor that accepts arguments.

## Constructor Overloading :-

A class can have multiple Constructors with different parameter lists.

Eg :- Car

Real-World Analogy = Construction Plan

### difference Between Constructor And Methods :-

→ Methods may have different names and will have a return type

→ Constructor's name must be same as the class

→ Methods are called explicitly and can be called any number of times while a Constructor's doesn't need to be called explicitly as it will be called implicitly during the object creation itself.

# Encapsulation & Abstraction

It involves bundling the data (variables) and methods (functions) that operate on the data into a single unit called "class".

## Real-World Analogy:

Eg:- Capsule

The medicine (data) is enclosed in a capsule (class) to protect it and ensure that it's released in a controlled manner when consumed (accessed).

Just as the capsule prevents the medicine from being exposed directly, encapsulation in Programming prevents the direct access of the data.

## Key Aspects of Encapsulation:-

-> Data Hiding

-> Controlled access

-> Modularity
[Organizing code into modular, self-contained units, making it easier to maintain and understand]

Example :- Banking System → Encapsulation

```
Public class BankAccount {

    Private String accountNo;
    Private double balance;        -> Data Hiding

    // Constructor to initialize account
    Public BankAccount (String accountNO, double balance) {
        this. accountNo = accountNO;
        this . balance = balance;
    }
    // Getter for accountNO
    Public String getAccountNumber() {
        return accountNO;
    }
    // Getter for balance
    Public double getBalance() {
        return balance;
    }
    // Setter to deposit amount
    Public void deposit (double amount) {
        if (amount > 0) {
            balance += amount;
        }
```

Controlled access using getters and setters

```
        else {
            S.O.P("Deposit amount must be +ve");
        }
    }
    // Setter to withdraw amount
    Public void withdraw (double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        } else {
            S.O.P("Insufficient amount(funds) or
                    invalid amount");
        }
    }
}
Public class Main {
    Public static void main (String[] args){
    // Creating a BankAccount object
    BankAccount account = new BankAccount("12345", 500.0);
    // Accessing data using methods
    account.deposit(200.0);
    account.withdraw(100.0);
    // Displaying account details using getters
    S.O.P("Account No.:" + account.getAccountNo());
    S.O.P("Balance:" + account.getBalance());
    }
}
```

## Abstraction

→ Hiding the unnecessary data.

Example :-

Banking Application itself

---

## Inheritance [Parent - Child relationship]

A Mechanism by which one object aquires all the properties and behaviors of a parent object.

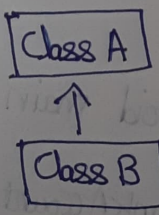
Use :-

→ Code Reusability

→ Method overriding (Runtime Polymorphism)

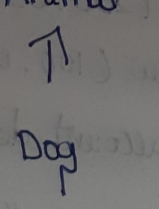
Types :-

a) Single :-

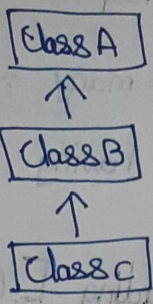



Single class inherits another class

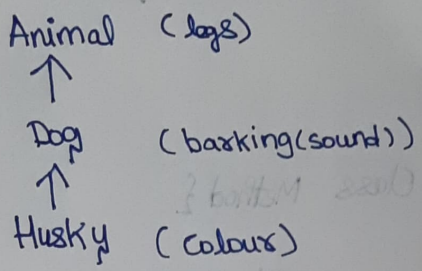
eg :-



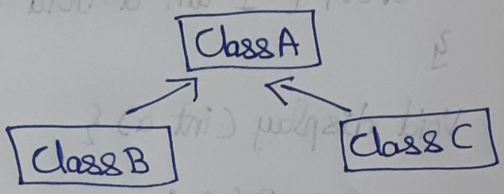
b) Multi-level :-



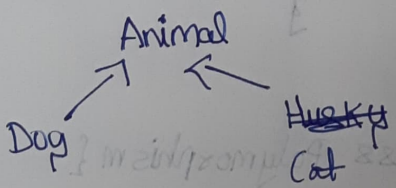

Eg :-




c) Hierarchial :-



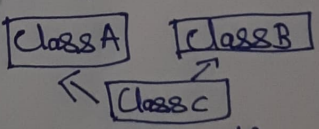

Eg :-




d) Hybrid :-

Combination of 1 or more types of inheritance

e) Multiple :-




To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

[! To avoid Compile time Error]

Decision of class C to choose between A or B.

# Polymorphism

[Poly = many + morphs = forms]

one method having different forms

* Method Overloading [Compile-time Polymorphism]

Eg:-

```
Class Method {
    Void display() {
        S.O.P("I am a void function");
    }
    Void display(int a) {
        S.O.P(a);
    }
}
Class Polymorphism {
    Public static void main(String[] args){
        Method obj = new Method();
        obj1.display();
        obj1.display(a:5);
    }
}
```

Real-time example:-

Use of min( )

arguments are varied

Method Overloading only occurs if the function has "different signatures".

Signature => i) No. of parameters passed
ii) ~~No~~ Return type.

---

## Method Overriding [Run-time Polymorphism]

-> Used when we need to change the specific implementation of a method over it's superclass.
-> Used in inheritance.

Eg:-

```
Class Vehicle {

    void run() {
        S.O.P("Vehicle is ~~running~~");   has 4 tyres
    }
}

Class Car extends Vehicle {

}

Class Bike extends Vehicle {

    void run() {
        S.O.P("Bike has 2 tyres");
    }
}

class main {
    Public static void main(String args[]) {

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

```
        obj.run()
    }
}
```

Output:

Vehicle has 4 tyres

Bike has 2 tyres

---

## ABSTRACT Classes & Abstract Methods

Abstract class:-

→ We Can't Create an object for an abstract class

→ This abstract class can be accessed with the help of inheritance.

Eg:-

```
abstract Class Computer {
    Void turnOn() {
        S.O.P ("Turning ON");
    }
}

Class HP extends Computer {
    Void turnOn() {          → Method Overriding
        S.O.P ("Turning ON-1");
    }
}
```

```
class Abstract {
    Public static void main(String[] args){
        HP ob= new HP();
        ob.turnOn();
    }
}
```

Output:-

Turning ON - 1

Output: (without Method overriding)

Turning ON.

Abstract Method:-

→ Created inside an abstract class

→ This class ~~should~~ must be overridden in other class which extends the abstract class.

→ This is only a function's declarative part and not function's definition part also.

→ The function which inherits this method will provide the function definition for it.

Eg:-

```
abstract class Computer {
    abstract void turnOn() {
        S.O.P ("Turning ON");
    }
}
```

```
Class HP extends Computer {
    Void turnON() {
        S.O.P ("Turning ON");
    }
}
Class Abstract {
    Public Static void main (String [] args){
        HP ob = new HP();
        ob.turnOn();
    }
}
```

Output :-

Turning ON

Note :-

→ Without Overriding there will be no output will be displayed on to the Screen.

# INTERFACES

→ Used in multiple inheritance allows us to have multiple parents

→ Also helps in abstraction

Note:

→ All Methods of an interface should always be an abstract class

Eg:

```
Interface Father {
    abstract void call();
    abstract void talk();
}

class Interface implements Father {
    Public void call(){
        S.O.P("Calling");
    }
    Public void talk(){
        S.O.P("Talking");
    }

    Public static void main(String[] args){
        Interface ob = new Interface();
        ob.call();
        ob.talk();
    }
}
```

# Multiple Inheritance

→ Can be done only by using "Abstract methods" and "interface classes" in Java.

→ Implements keyword is used to inherit the Parent classes property.

Eg:

```
interface Father {
    abstract void call();
    ,,      ,, talk();
}
interface Mother {
    abstract void call();
    ,,      ,, talk();
}
Class Son implements Father, Mother {
    Public void call() {
        S.O.P ("Calling");
    }
    Public void talk() {
        S.O.P ("Talking");
    }
```

```
Public static void main (String[] args){
    Son ob = new Son();
    ob.callO();
    ob.talk();
  }
}
```

Output :

Calling
Talking

---

## Static Keyword

=> Used to access a class without any object Creation or Instantiation.

=> Memory allocation is better because there is no new object Created.

Public static void main (String[] args)

↓

Why we give static here?

Because if a Compiler needs to access the main function it has to create an object to do so and to avoid this object Creation Scenario "static" keyword is Used, so that the Compiler can easily access the main function(method).

## Static Advantage over a Variable :-

→ Same as method's advantage we don't need to Create an object Separately for accessing.

→ Same instance Variable is used reducing the memory usage. [Act's as a Common Variable]

[2.20.00] ↓ Timeline

Eg :-

Father age, Daughter age

If we give static Father age in father Class and then if we access the age again after changing the age after initialization then the Output will be the changed age in the main method and not the age in the Father Class.

---

## Final Keyword

A Modifier used to restrict the usage of Variables, Methods and Classes

### Final Variable:

A Variable declared with "final" Cannot be reassigned after it's initialization.

eg :-

```
final int a = 100;
```

## Finalize() Method:-

=> Starting from Java 9 this method is deprecated.

=> This method is called by "Garbage Collector" before an object is destroyed.

=> This has lot of disadvantages that the method will be called or not at all in a timely manner.

=> this is an outdated mechanism has manual cleanup's are introduced nowadays.

## Finally Block:-

=> Used after 'try' 'catch' block in exception handling.

=> It will always be executed regardless of the execution of "try" "Catch" block

Eg:-

```
try {
    int data=10/0;
}
Catch (ArithmeticException e){
S.O.P(e);
} finally {
S.O.P("Finally block executed");
}
```

Output:- Arithmetic Exception
Finally block executed

# Super Keyword

⇒ Used to refer the "Parent" or "Superclass" of the current object.

Primarily used in 3 contexts:-

i) Accessing Parent class variable's :-

Eg:-

Accessing Father's ~~class~~ gender in Father class directly.

ii) Accessing Parent class method's :-

Eg:-

Accessing Father class's void print() method

iii) Accessing Parent class Constructor:-

Whole Parent class constructor() itself.

↓

Eg:-

```
Class Parent {
    Parent() {
        System.out.P ("Parent class Constructor");
    }
}
Class Child extends Parent {
    Child() {
        Super();
        S.O.P("Child class Constructor");
    }
}
```

```
Public Class Main{
    Public static void main(string[] args){
        Child child = new child();
    }
}
```

Output :-

Parent class Constructor
Child " "

---

## This Keyword

Used to allow an argument Passed to access an instance Variable.

Eg :-

```
Class Fathers{
    Char gender; -> instance Variable
    int age;
    Void print(char gender, int age){
        this.gender = gender;
        this.age = age;
    }
}
```

Arguments

Note! :-

If not used the arguments will not access the instance Variable instead it will assign itself to the instance Variable.

gender = gender;

This will print a null character.