OOPS

Object-Oriented Programming (OOP) is a programming methodology or paradigm used to design a program using **classes** and **objects**.
It makes software development and maintenance easier by organizing the program into reusable and real-world related components.

Class

A **class** is a user-defined data type that defines the **properties (data members)** and **functions (methods)** of an object.
A class is only a **logical representation** of data.
It does **not occupy memory** until an object of the class is created.

Example:
Human being is a class.
Body parts are its properties, and actions performed by body parts are its functions.

Java Syntax (Class):

```
class Student {
    int id;        // data member
    int mobile;
    String name;

    int add(int x, int y) {    // member function
        return x + y;
    }
}
```

Object

An **object** is a **run-time entity**.
It is an **instance of a class**.
An object can represent a **person, place, or any real-world entity**.
An object can access both **data members** and **member functions** of a class.

Java Syntax (Object):

```
Student s = new Student();
```

Note:
When an object is created using the new keyword in Java:

- Memory is allocated in the **heap**
- The reference variable is stored in the **stack**

Java objects are always created using the new keyword.

Inheritance

Inheritance is a process in which one class acquires the **properties (data members)** and **behaviors (methods)** of another class automatically.
It helps in **code reusability**, **extending**, and **modifying** existing class features.

In **Java**:

- The class whose properties are inherited is called the **parent class (superclass)**.
- The class that inherits the properties is called the **child class (subclass)**.
- The child class is a specialized version of the parent class.

Java Syntax (Inheritance)

```
class ChildClass extends ParentClass {
    // additional properties or methods
}
```

Types of Inheritance in Java

1. Single Inheritance

When one class inherits another single class.

```
class Parent {
    int a = 10;
}

class Child extends Parent {
    int b = 20;
}
```

2. Multilevel Inheritance

When a class is derived from another derived class.

```
class GrandParent {
    int a = 10;
}

class Parent extends GrandParent {
    int b = 20;
```

```
}

class Child extends Parent {
    int c = 30;
}
```

3. Hierarchical Inheritance

When multiple classes inherit from a single parent class.

```
class Parent {
    int a = 10;
}

class Child1 extends Parent {
}

class Child2 extends Parent {
}
```

4. Multiple Inheritance

Java **does not support multiple inheritance using classes** to avoid ambiguity.
It can be achieved using **interfaces**.

```
interface A {
    void show();
}

interface B {
    void display();
}

class Child implements A, B {
    public void show() {}
    public void display() {}
}
```

5. Hybrid Inheritance

Hybrid inheritance is a combination of **two or more types of inheritance**.
Java supports hybrid inheritance **using interfac**

Note:

Java uses the extends keyword for class inheritance and implements keyword for interface inheritance.

Encapsulation

Encapsulation is the process of **wrapping data (variables)** and **methods (functions)** into a single unit called a **class**.
In encapsulation, **data is not accessed directly**. It is accessed using **methods** defined inside the class.

In simple words:

- Class variables are kept **private**
- **Public getter and setter methods** are used to access and modify the data
- This provides **data hiding** and **security**

Data Hiding

Data hiding means restricting direct access to class data to reduce misuse and dependency.
In Java, this is achieved using **access modifiers** like private and protected.

Java Example (Encapsulation)

```
class Student {
    private int id;        // private data member
    private String name;

    // Getter method
    public int getId() {
        return id;
    }

    // Setter method
    public void setId(int id) {
        this.id = id;
    }

    // Getter method
    public String getName() {
        return name;
    }

    // Setter method
```

```
    public void setName(String name) {
        this.name = name;
    }
}
```

Advantages of Encapsulation

- Provides **data security**
- Improves **code maintainability**
- Makes the program **flexible and reusable**
- Helps in **data hiding**

Abstraction

Abstraction means showing only the **important details** and hiding the **unnecessary details** of a real-world problem.
It helps us focus on **what an object does** instead of **how it does it**.

In simple words:

- We create a **general model** of a real-life problem
- Only essential properties and operations are defined
- Similar real-world problems can use the **same abstract solution**

In Java, abstraction is achieved using:

- **Abstract classes**
- **Interfaces**

Java Example (Abstraction using Abstract Class)

```
abstract class Vehicle {
    abstract void start();   // abstract method
}

class Bike extends Vehicle {
    void start() {
        System.out.println("Bike starts with key");
    }
}
```

Data Binding

Data binding is the process of **connecting application UI with business logic**.
When a change is made in the **business logic**, it is **automatically reflected in the UI**.

Advantages of Abstraction

- Reduces complexity
- Improves code reusability
- Makes the system easier to understand
- Helps in solving real-world problems efficiently

Polymorphism

Polymorphism means the ability of an object to take **many forms**.
The same method name can behave **differently** depending on the object that is calling it.

In simple words:

- **Poly** means *many*
- **Morphism** means *forms*
- One interface, **multiple implementations**

Types of Polymorphism

1. Compile-Time Polymorphism (Static Polymorphism)

This type of polymorphism is resolved at **compile time**.
It is achieved using **method overloading**.

Method Overloading

Method overloading means having **multiple methods with the same name** but with:

- Different number of parameters
- Different type of parameters

(Return type alone cannot change method overloading in Java)

Java Example (Method Overloading)

```
class Add {
    int add(int a, int b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }
}

class Main {
    public static void main(String[] args) {
        Add obj = new Add();
        int res1 = obj.add(2, 3);
        int res2 = obj.add(2, 3, 4);
        System.out.println(res1 + " " + res2);
    }
}
```

Output:

```
5 9
```

2. Runtime Polymorphism (Dynamic Polymorphism)

Runtime polymorphism is resolved at **runtime**.
It is achieved using **method overriding**.

Method Overriding

Method overriding occurs when:

- A child class provides its own implementation of a method already present in the parent class
- Method call depends on the **object type**, not the reference type

Java Example (Method Overriding)

```
class BaseClass {
    void show() {
        System.out.println("Base class method");
    }
}
```

```
class DerivedClass extends BaseClass {
    void show() {
        System.out.println("Derived class method");
    }
}

class Main {
    public static void main(String[] args) {
        BaseClass obj;
        obj = new DerivedClass();
        obj.show();   // calls Derived class method
    }
}
```

Advantages of Polymorphism

- Improves code flexibility
- Supports code reusability
- Makes program easier to maintain
- Provides dynamic behavior at runtime

Constructor

A **constructor** is a special method that is **automatically called when an object is created**.
It is mainly used to **initialize data members** of a class.

In **Java**:

- Constructor name is **same as the class name**
- Constructors **do not have a return type**

Types of Constructors in Java

1. Default Constructor

A constructor with **no parameters** is called a default constructor.
It is called automatically when an object is created without arguments.

```
class Student {
    int id;

    Student() {        // default constructor
```

```
        id = 0;
    }
}
```

2. Parameterized Constructor

A constructor that has **parameters** is called a parameterized constructor.
It is used to initialize objects with **different values**.

```
class Student {
    int id;

    Student(int i) {   // parameterized constructor
        id = i;
    }
}
```

3. Copy Constructor

Java does **not have a built-in copy constructor** like C++.
However, copying can be done by **passing an object as a parameter** to a constructor.

```
class Go {
    int x;

    Go(int a) {          // parameterized constructor
        x = a;
    }

    Go(Go obj) {         // copy constructor (user-defined)
        x = obj.x;
    }
}

class Main {
    public static void main(String[] args) {
        Go a1 = new Go(20);   // parameterized constructor
        Go a2 = new Go(a1);  // copy constructor
        System.out.println(a2.x);
    }
}
```

Output:

```
20
```

Key Points

- Constructor is called automatically
- Used to initialize objects
- Same name as class
- No return type
- Supports constructor overloading

Destructor

A **destructor** is used to **destroy an object** and **free resources** when the object is no longer needed.
It works **opposite to a constructor** and is **called automatically**.

Destructor in Java

Java **does not have destructors** like C++.
In Java, memory management is handled automatically by the **Garbage Collector**.

To perform cleanup actions, Java provides a special method called **finalize()** (now deprecated, but used for understanding).

Java Example (Destructor-like behavior)

```
class A {

    // Constructor
    A() {
        System.out.println("Constructor in use");
    }

    // Destructor-like method
    protected void finalize() {
        System.out.println("Destructor in use");
    }
}

class Main {
    public static void main(String[] args) {
        A obj = new A();
        obj = null;       // object becomes eligible for garbage collection
        System.gc();      // request garbage collection
    }
```

```
}
```

Output (Possible Output)

```
Constructor in use
Destructor in use
```

Important Points

- Java does not support destructors explicitly
- Garbage Collector automatically frees memory
- finalize() is used to understand object destruction
- Constructor is used for initialization
- Destructor-like behavior is automatic in Java

this Keyword

this is a keyword in Java that refers to the **current object** of the class.
It is used to differentiate between **instance variables** and **local variables** and to represent the **current class object**.

Uses of this Keyword in Java

1. Referring to Current Class Instance Variable

Used when instance variables and method parameters have the same name.

```
class Student {

    int id;


    Student(int id) {

        this.id = id;   // refers to instance variable

    }

}
```

2. Passing Current Object as a Parameter

The current object can be passed to another method using this.

```
class Test {
    void display(Test obj) {
        System.out.println("Method called");
    }

    void show() {
        display(this);   // passing current object
    }
}
```

3. Calling Another Constructor of the Same Class

this() is used to call another constructor in the same class.

```
class Example {
    Example() {
        System.out.println("Default constructor");
    }

    Example(int x) {
        this();   // calls default constructor
        System.out.println("Parameterized constructor");
    }
}
```

Java Example Similar to Given C++ Code

```
class Node {
    int data;
```

```
    Node next;


    Node(int x) {
        this.data = x;
        this.next = null;
    }
}
```

Key Points

- this refers to the current object
- Used to access instance variables
- Used to pass current object
- Used to call another constructor

Friend Function

In **C++**, a **friend function** is a non-member function that can access the **private and protected members** of a class.

Friend Function in Java

Java does NOT support friend functions.

Java follows **strict encapsulation**, so:

- A **non-member function cannot access private data**
- There is **no friend keyword** in Java

How Java Achieves Similar Functionality

In Java, similar behavior is achieved using:

- **Public methods**
- **Getter methods**
- **Package-level access (default access)**

Instead of friend functions, Java allows access through **methods inside the class**.

Java Equivalent Example

```
class A {
    private int a = 2;
    private int b = 4;

    // public method to access private data
    public int mul() {
        return a * b;
    }
}

class Main {
    public static void main(String[] args) {
        A obj = new A();
        int res = obj.mul();
        System.out.println(res);
    }
}
```

Output:

```
8
```

Key Points

- Java does not support friend functions
- Private data is accessed using public methods
- This maintains data security and encapsulation
- Java prefers object-oriented access instead of friend functions

Aggregation

Aggregation is a special form of **association** where one class **contains the reference of another class**.
It represents a **HAS-A relationship**.

In aggregation:

- Both classes can **exist independently**
- One object uses another object as a part

Java Example (Aggregation)

```
class Address {
    String city;
    String state;

    Address(String city, String state) {
        this.city = city;
        this.state = state;
    }
}

class Student {
    int id;
    String name;
    Address address;   // HAS-A relationship (Aggregation)

    Student(int id, String name, Address address) {
        this.id = id;
        this.name = name;
        this.address = address;
```

```
    }
}

class Main {
    public static void main(String[] args) {
        Address addr = new Address("Pune", "Maharashtra");
        Student s = new Student(1, "Rahul", addr);
    }
}
```

Key Points

- Aggregation represents **HAS-A relationship**
- One class contains another class object
- Classes can exist independently
- Used for code reusability

Virtual Function (Java Concept)

In **Java**, all **non-static methods are virtual by default**.
This means Java automatically supports **runtime polymorphism** without using the virtual keyword.

A **virtual function** allows the **derived class method** to be called **at runtime**, even when the object is accessed using a **base class reference**.

Meaning in Simple Words

- Method is defined in **parent class**
- Same method is **overridden in child class**
- Method call is decided **at runtime**
- Object type decides which method is executed

Key Points (Java)

1. All instance methods in Java are **virtual by default**
2. Static methods **cannot be overridden**
3. Constructors **cannot be virtual**
4. Method call depends on **object**, not reference

Java Example (Equivalent of Virtual Function)

```
class Base {
    void print() {
        System.out.println("print base class");
    }

    void show() {
        System.out.println("show base class");
    }
}

class Derived extends Base {
    void print() {
        System.out.println("print derived class");
    }

    void show() {
        System.out.println("show derived class");
    }
}

class Main {
```

```
    public static void main(String[] args) {
        Base bptr;
        Derived d = new Derived();
        bptr = d;

        bptr.print();   // runtime binding
        bptr.show();    // runtime binding
    }
}
```

Output

```
print derived class
show derived class
```

Important Difference from C++

- Java does **not** need the virtual keyword
- Method binding happens **automatically at runtime**
- Java does not support virtual constructors
- Java uses **dynamic method dispatch**

Pure Virtual Function (Java Equivalent)

In **Java**, a **pure virtual function** is called an **abstract method**.
A class that contains an abstract method is called an **abstract class**.

Meaning in Simple Words

- An abstract method has **no body**
- It is declared in the **base class**
- The **derived class must override** the method
- Objects of the abstract class **cannot be created**

- Used to achieve **runtime polymorphism**

Key Points

1. Abstract methods do **not perform any task**
2. Abstract class acts as a **blueprint**
3. Child class must implement all abstract methods
4. Abstract class reference can point to child object

Java Syntax (Abstract Method)

```
abstract void show();
```

Java Example (Equivalent of Pure Virtual Function)

```
abstract class Base {
    abstract void show();   // abstract method
}

class Derived extends Base {
    void show() {
        System.out.println("You can see me!");
    }
}

class Main {
    public static void main(String[] args) {
        Base bptr;
        Derived d = new Derived();
```

```
        bptr = d;

        bptr.show();

    }

}
```

Output

```
You can see me!
```

Important Points

- Java uses abstract instead of pure virtual
- Abstract methods have no implementation
- Abstract classes cannot be instantiated
- Supports runtime polymorphism

Abstract Classes

In **Java**, an **abstract class** is a class that **cannot be instantiated** and is used as a **base class**.
It may contain **abstract methods** (methods without body) and **non-abstract methods**.

An abstract method must be **implemented by the derived class**.

Key Points

- Abstract class is declared using the abstract keyword
- Abstract method has **no body**
- Object of abstract class **cannot be created**
- Used to achieve **abstraction and runtime polymorphism**

Java Example (Abstract Class)

```
abstract class Shape {

    abstract void draw();   // abstract method

}
```

```
class Rectangle extends Shape {
    void draw() {
        System.out.println("Rectangle");
    }
}

class Square extends Shape {
    void draw() {
        System.out.println("Square");
    }
}

class Main {
    public static void main(String[] args) {
        Rectangle rec = new Rectangle();
        Square sq = new Square();

        rec.draw();
        sq.draw();
    }
}
```

Output

```
Rectangle
Square
```

Important Points

- Abstract class provides a common structure
- Child classes must implement abstract methods
- Helps in code reusability
- Improves program design

Namespaces (Java Equivalent)

Java does **not use namespaces** like C++.
Instead, Java uses **packages** to achieve the same purpose.

Meaning in Simple Words

- Namespace/Package is used to **avoid name conflicts**
- It provides a **logical grouping of classes**
- It defines the **scope** of classes, interfaces, and methods
- Helps in **code organization and reusability**

Key Points (Java Packages)

1. Packages prevent **naming conflicts**
2. Classes with the same name can exist in **different packages**
3. Java has a built-in package java.lang
4. User-defined packages can also be created

Java Example (Package)

```
package add;


public class Add {

    int a = 5;

    int b = 5;


    public int add() {
```

```
        return a + b;
    }
}

import add.Add;

class Main {
    public static void main(String[] args) {
        Add obj = new Add();
        int res = obj.add();
        System.out.println(res);
    }
}
```

Output

```
10
```

Important Points

- Java uses **packages instead of namespaces**
- import keyword is used to access package classes
- Avoids ambiguity in large programs
- Improves modularity and maintenance

Access Specifiers

Access specifiers are used to control **how variables and methods can be accessed** from outside a class.
They provide **data security** and support **encapsulation**.

In **Java**, there are **three important access specifiers**:

1. Private

- Members declared as private are accessible **only inside the same class**
- They **cannot be accessed** outside the class

```
class Test {

   private int x = 10;

}
```

2. Public

- Members declared as public are accessible **from anywhere**
- No access restriction

```
class Test {

   public int x = 10;

}
```

3. Protected

- Members declared as protected are accessible:
    - Within the **same package**
    - In **child classes** (even in different packages)
- Mostly used in **inheritance**

```
class Test {

   protected int x = 10;

}
```

Key Points

- Access specifiers improve data security
- Help in encapsulation
- Widely used in object-oriented programming
- Important for inheritance concepts

Important OOP Concepts (Java Explanation)

Memory Release (delete in C++ vs Java)

- Java **does not use delete or delete[]**
- Memory is **automatically managed** by the **Garbage Collector**
- Objects are removed when they are **no longer referenced**

Virtual Inheritance

- Java does **not support virtual inheritance**
- Java avoids duplication problem using **interfaces**
- Only **one copy** of interface methods is used

Function Overloading

Function overloading means having **multiple methods with the same name** but:

- Different number of parameters
- Different type of parameters

It is an example of **compile-time (static) binding**.

```
class Add {
    int add(int a, int b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

Operator Overloading

- Java **does not support operator overloading**
- The only exception is + operator (used for String concatenation)

Method Overloading vs Method Overriding

Method Overloading

- Same method name
- Different parameters
- Same class
- Compile-time binding

Method Overriding

- Same method name
- Same parameters and return type
- Parent–child class relationship
- Runtime binding

Java Example (Overriding)

```
class Parent {
    void show() {
        System.out.println("Parent class");
    }
}


class Child extends Parent {
    void show() {
        System.out.println("Child class");
    }
}
```

Key Differences

Overloading	Overriding
Compile-time binding	Runtime binding
Same class	Parent-child class
Different parameters	Same parameters
Increases readability	Provides runtime polymorphism