Java OOP(Object Oriented Programming) Concepts

Last Updated : 14 Apr, 2025

Java Object-Oriented Programming (OOPs) is a fundamental concept in Java that every developer must understand. It allows developers to structure code using **classes and objects**, making it more modular, reusable, and scalable.

The core idea of **OOPs** is to bind data and the functions that operate on it, preventing unauthorized access from other parts of the code. Java strictly follows the DRY (Don't Repeat Yourself) Principle, ensuring that common logic is written once (e.g., in parent classes or utility methods) and reused throughout the application. This makes the code:

- **Easier to maintain:** Changes are made in one place.
- **More organized:** Follows a structured approach.
- **Easier to debug and understand**: Reduces redundancy and improves readability.

In this article, we will explore how **OOPs works in Java** using classes and objects. We will also dive into its **four main pillars of OOPs** that are, **Abstraction, Encapsulation, Inheritance**, and **Polymorphism** with examples.

What is OOPs and Why Do We Use it?

OOPS stands for **Object-Oriented Programming** System. It is a programming approach that organizes code into objects and classes and makes it more structured and easy to manage. A class is a blueprint that defines properties and behaviors, while an object is an instance of a class representing real-world entities.

Example:

```java
// Use of Object and Classes in Java
import java.io.*;

class Numbers {
    // Properties
    private int a;
    private int b;

    // Setter methods
    public void setA(int a) { this.a = a; }
    public void setB(int b) { this.b = b; }

    // Methods
    public void sum() { System.out.println(a + b); }
    public void sub() { System.out.println(a - b); }

    public static void main(String[] args)
    {
        Numbers obj = new Numbers();

        // Using setters instead of direct access
        obj.setA(1);
        obj.setB(2);

        obj.sum();
        obj.sub();
    }
}
```

Output

```
3
-1
```

It is a simple example showing a class Numbers containing two variables which can be accessed and updated only by instance of the object created.

Java Class

A Class is a **user-defined blueprint** or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times. This includes classes for objects occurring more than once in your code. In general, class declarations can include these components in order:

- **Modifiers**: A class can be public or have default access (Refer to this for details).
- **Class name:** The class name should begin with the initial letter capitalized by convention.
- **Body:** The class body is surrounded by braces, { }.

Java Object

An Object is a basic unit of Object-Oriented Programming that represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. The objects are what perform your code, they are the part of your code visible to the viewer/user. An object mainly consists of:

- **State**: It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity**: It is a unique name given to an object that enables it to interact with other objects.
- **Method**: A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to **reuse** the code without retyping it, which is why they are considered **time savers**. In Java, every method must be part of some class, which is different from languages like C, C++, and Python.

Example:

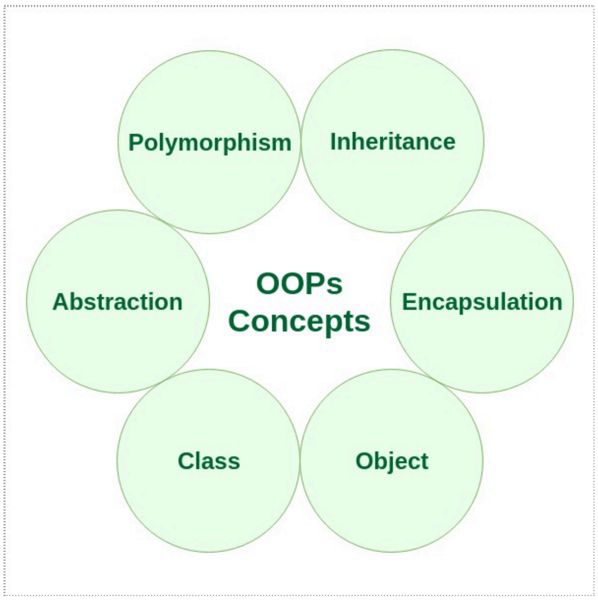
```java
// Java Program to demonstrate
// Use of Class and Objects

// Class Declared
public class Employee {
    // Instance variables (non-static)
    private String name;
    private float salary;

    // Constructor
    public Employee(String name, float salary) {
        this.name = name;
        this.salary = salary;
    }

    // getters method
    public String getName() { return name; }
    public float getSalary() { return salary; }

    // setters method
    public void setName(String name) { this.name = name; }
    public void setSalary(float salary) { this.salary = salary; }

    // Instance method
    public void displayDetails() {
        System.out.println("Employee: " + name);
        System.out.println("Salary: " + salary);
    }

    public static void main(String[] args) {
        Employee emp = new Employee("Geek", 10000.0f);
        emp.displayDetails();
    }
}
```

Output

```
Employee: Geek
Salary: 10000.0
```

Note: For more information, please refer to the article - Classes and Object.

The below diagram demonstrates the Java OOPs Concepts



Method and Method Passing

A method is a collection of statements that perform specific tasks and return a result to the caller. It can be declared with or without arguments, depending on the requirements. A method can take input values, perform operations, and return a result.

Example:

```java
// Class Method and Method Passing
class Student {
    private int id;
    private String name;

    // Constructor for initialization
    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // method demonstrating parameter passing
    public void printStudent(String header) {
        System.out.println(header);
```

```java
            System.out.println("ID: " + getId());
            System.out.println("Name: " + getName());
        }

        // Getter methods
        public int getId() { return id; }
        public String getName() { return name; }
    }

    class Main {
        public static void main(String[] args) {
            // Proper initialization
            Student obj = new Student(28, "Geek");
            // Method with parameter
            obj.printStudent("Student Details:");
        }
    }
```

Output

```
Student Details:
ID: 28
Name: Geek
```

4 Pillars of Java OOPs Concepts

1. Abstraction

Data Abstraction is the property by virtue of which **only the essential details are displayed to the user.** The trivial or non-essential units are not displayed to the user. Data Abstraction may also be defined as the process of identifying only the required characteristics of an object, ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the object.

Real-life Example: Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the car speed or applying brakes will stop the car, but he does not know how on pressing the accelerator, the speed is actually increasing. He does not know about the inner mechanism of the car or the implementation of the accelerators, brakes etc. in the car. This is what abstraction is.

Note: In Java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

Example:

```java
// Abstract class representing a Vehicle (hidi
implementation details)
abstract class Vehicle {
    // Abstract methods (what it can do)
    abstract void accelerate();
    abstract void brake();

    // Concrete method (common to all vehicles)
    void startEngine() {
        System.out.println("Engine started!");
    }
}

// Concrete implementation (hidden details)
class Car extends Vehicle {
    @Override
    void accelerate() {
        System.out.println("Car: Pressing gas
pedal...");
        // Hidden complex logic: fuel injection,
gear shifting, etc.
    }

    @Override
    void brake() {
        System.out.println("Car: Applying brakes...");
        // Hidden logic: hydraulic pressure, brake pads, etc.
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        myCar.startEngine();
        myCar.accelerate();
        myCar.brake();
    }
}
```

Note: To learn more about the Abstraction refer to the **Abstraction in Java** article

2. Encapsulation

It is defined as the **wrapping up of data under a single unit.** It is the mechanism that binds together the code and the data it manipulates. Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

- Technically, in encapsulation, the variables or the data in a class is hidden from any other class and can be accessed only through any member function of the class in which they are declared.
- In encapsulation, the data in a class is hidden from other classes, which is similar to what **data-hiding** does. So, the terms "encapsulation" and "data-hiding" are used interchangeably.
- Encapsulation can be achieved by declaring all the variables in a class as private and writing public methods in the class to set and

get the values of the variables.

Example:

```java
// Encapsulation using private modifier

class Employee {
    // Private fields (encapsulated data)
    private int id;
    private String name;

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

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

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

    public String getName() {
        return name;
    }
}

public class Main {
    public static void main(String[] args) {
        Employee emp = new Employee();

        // Using setters
        emp.setId(101);
        emp.setName("Geek");

        // Using getters
        System.out.println("Employee ID: " +
emp.getId());
        System.out.println("Employee Name: " +
emp.getName());
    }
}
```

Output

```
Employee ID: 101
Employee Name: Geek
```

Note: To learn more about topic refer to [Encapsulation in Java] article.

3. Inheritance

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class. We are achieving inheritance by using **extends** keyword. Inheritance is also known as "**is-a**" relationship.

Let us discuss some frequently used important terminologies:

- **Superclass:** The class whose features are inherited is known as superclass (also known as base or parent class).
- **Subclass:** The class that inherits the other class is known as subclass (also known as derived or extended or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Example:

```java
// Superclass (Parent)
class Animal {
    void eat() {
        System.out.println("Animal is eating...");
    }

    void sleep() {
        System.out.println("Animal is sleeping...");
    }
}

// Subclass (Child) - Inherits from Animal
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();

        // Inherited methods (from Animal)
        myDog.eat();
        myDog.sleep();

        // Child class method
        myDog.bark();
    }
}
```

Output

```
Animal is eating...
Animal is sleeping...
Dog is barking!
```

Note: To learn more about topic refer to [Inheritance in Java](#) article.

4. Polymorphism

It refers to the ability of object-oriented programming languages **to differentiate between entities with the same name efficiently**. This is done by Java with the help of the signature and declaration of these entities. The ability to appear in many forms is called polymorphism.

Example:

```
sleep(1000) //millis
sleep(1000,2000) //millis,nanos
```

Types of Polymorphism

Polymorphism in Java is mainly of 2 types as mentioned below:

1. Method Overloading
2. Method Overriding

Method Overloading and Method Overriding

1. Method Overloading: Also, known as **compile-time polymorphism**, is the concept of Polymorphism where more than one method share the same name with different signature(Parameters) in a class. The return type of these methods can or cannot be same.

2. Method Overriding: Also, known as run-time polymorphism, is the concept of Polymorphism where method in the child class has the same name, return-type and parameters as in parent class. The child class provides the implementation in the method already written.

Below is the implementation of both the concepts:

```
// Java Program to Demonstrate
// Method Overloading and Overriding

// Parent Class
class Parent {
    // Overloaded method (compile-time polymorphism)
    public void func() {
        System.out.println("Parent.func()");
    }

    // Overloaded method (same name, different
parameter)
```

```java
    public void func(int a) {
        System.out.println("Parent.func(int): " + a);
    }
}

// Child Class
class Child extends Parent {
    // Overrides Parent.func(int) (runtime polymorphism)
    @Override
    public void func(int a) {
        System.out.println("Child.func(int): " + a);
    }
}

public class Main {
    public static void main(String[] args) {
        Parent parent = new Parent();
        Child child = new Child();
        // Dynamic dispatch
        Parent polymorphicObj = new Child();

        // Method Overloading (compile-time)
        parent.func();
        parent.func(10);

        // Method Overriding (runtime)
        child.func(20);

        // Polymorphism in action
        polymorphicObj.func(30);
    }
}
```

Output

```
Parent.func()
Parent.func(int): 10
Child.func(int): 20
Child.func(int): 30
```

Advantage of OOPs over Procedure-Oriented Programming Language

Object-oriented programming (OOP) offers several key advantages over procedural programming:

- By using objects and classes, you can create reusable components, leading to less duplication and more efficient development.
- It provides a clear and logical structure, making the code easier to understand, maintain, and debug.
- OOP supports the DRY (Don't Repeat Yourself) principle.This principle encourages minimizing code repetition, leading to cleaner, more maintainable code. Common functionalities are placed in a single location and reused, reducing redundancy.
- By reusing existing code and creating modular components, OOP allows for quicker and more efficient application development

Disadvantages of OOPs

- OOP has concepts like classes, objects, inheritance etc. For beginners, this can be confusing and takes time to learn.
- If we write a small program, using OOP can feel too heavy. We might have to write more code than needed just to follow the OOP structure.
- The code is divided into different classes and layers, so in this, finding and fixing bugs can sometimes take more time.
- OOP creates a lot of objects, so it can use more memory compared to simple programs written in a procedural way.

Conclusion

The Object Oriented Programming (OOPs) concept in Java is a powerful way to organize and write code. It uses key ideas like classes, objects, inheritance, polymorphism, encapsulation, and abstraction to create flexible and reusable code.