**Introduction to OOP**

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes to structure software programs. It aims to implement real-world entities like inheritance, hiding, polymorphism, and more in programming. Let’s explore the core concepts of OOP:

**Core Concepts of OOP**

1. **Class**
   * A class is a blueprint for creating objects. It defines a set of attributes and methods that the created objects will have.
   * Example:

**Java**

public class Car {

String color;

String model;

void drive() {

System.out.println("Car is driving");

}

}

1. **Object**
   * An object is an instance of a class. It represents a real-world entity with a state and behavior.
   * Example:

**Java**

Car myCar = new Car();

myCar.color = "Red";

myCar.model = "Toyota";

myCar.drive(); // Output: Car is driving

1. **Encapsulation**
   * Encapsulation is the wrapping up of data and methods into a single unit (class). It restricts direct access to some of the object’s components, which can prevent the accidental modification of data.
   * Example:

**Java**

public class Car {

private String color;

public String getColor() {

return color;

}

public void setColor(String color) {

this.color = color;

}

}

1. **Inheritance**
   * Inheritance allows a new class to inherit properties and methods from an existing class. This promotes code reusability.
   * Example:

**Java**

public class Vehicle {

void start() {

System.out.println("Vehicle is starting");

}

}

public class Car extends Vehicle {

void drive() {

System.out.println("Car is driving");

}

}

1. **Polymorphism**
   * Polymorphism allows methods to do different things based on the object it is acting upon, even though they share the same name.
   * Example:

**Java**

public class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

public class Dog extends Animal {

void sound() {

System.out.println("Dog barks");

}

}

public class Cat extends Animal {

void sound() {

System.out.println("Cat meows");

}

}

1. **Abstraction**
   * Abstraction is the concept of hiding the complex implementation details and showing only the necessary features of an object.
   * Example:

**Java**

abstract class Animal {

abstract void sound();

}

public class Dog extends Animal {

void sound() {

System.out.println("Dog barks");

}

}

**Benefits of OOP**

* **Modularity**: The source code for an object can be written and maintained independently of the source code for other objects.
* **Reusability**: Objects can be reused across programs.
* **Pluggability and Debugging Ease**: If a particular object turns out to be problematic, it can be removed and replaced with a better object.

**OOP - Polymorphism**

Polymorphism is one of the core concepts of Object-Oriented Programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different underlying forms (data types). Let’s explore the two main types of polymorphism in Java: **compile-time polymorphism** and **runtime polymorphism**.

**Compile-Time Polymorphism (Static Polymorphism)**

Compile-time polymorphism is achieved through method overloading. Method overloading allows multiple methods in the same class to have the same name but different parameters (different type, number, or both).

**Example:**

**Java**

public class MathUtils {

// Method overloading

public int add(int a, int b) {

return a + b;

}

public double add(double a, double b) {

return a + b;

}

public int add(int a, int b, int c) {

return a + b + c;

}

public static void main(String[] args) {

MathUtils math = new MathUtils();

System.out.println(math.add(2, 3)); // Output: 5

System.out.println(math.add(2.5, 3.5)); // Output: 6.0

System.out.println(math.add(1, 2, 3)); // Output: 6

}

}

**Runtime Polymorphism (Dynamic Polymorphism)**

Runtime polymorphism is achieved through method overriding. Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This is typically used in the context of inheritance.

**Example:**

**Java**

class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

void sound() {

System.out.println("Dog barks");

}

}

class Cat extends Animal {

@Override

void sound() {

System.out.println("Cat meows");

}

}

public class Main {

public static void main(String[] args) {

Animal myAnimal = new Animal();

Animal myDog = new Dog();

Animal myCat = new Cat();

myAnimal.sound(); // Output: Animal makes a sound

myDog.sound(); // Output: Dog barks

myCat.sound(); // Output: Cat meows

}

}

In this example, the sound method is overridden in the Dog and Cat classes. [The method that gets called is determined at runtime based on the object type](https://stackify.com/oop-concept-polymorphism/).

**Benefits of Polymorphism**

* **Flexibility**: Polymorphism allows for code that can work with objects of different types and classes.
* **Maintainability**: It makes the code easier to maintain and extend.
* **Reusability**: Polymorphic code can be reused across different parts of the application.