**Object-Oriented Programming (OOP) Concepts**

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain both data (attributes) and methods (functions or procedures) that operate on the data. The goal of OOP is to structure code in a way that is more manageable, reusable, and easier to maintain.

**Object**

An **object** is a fundamental unit in OOP that represents a real-world entity or concept. An object has:

* **State**: The data or attributes that define the characteristics of the object.
* **Behavior**: The methods or functions that define how the object can act or interact with other objects.
* **Identity**: The unique distinction of an object, often represented by a memory address or a reference in the system.

**Characteristics of an Object**

1. **State**: The state of an object refers to the specific data or attributes that describe the current condition of the object. For example, a car object might have attributes such as color, model, and speed.
2. **Behavior**: The behavior of an object refers to the operations or methods it can perform. For a car object, methods could include accelerate() or brake().
3. **Identity**: Every object has a unique identity, which distinguishes it from other objects. This is usually handled by the memory address or reference.
4. **Responsibility**: An object has responsibilities, which are tasks it is supposed to perform. An object is responsible for maintaining its own state and providing methods to interact with it.

**Major Pillars of Object-Oriented Programming in Java**

1. **Abstraction**
   * Abstraction is the process of hiding the complex implementation details and exposing only the essential functionality. This makes the system easier to use.
   * In Java, this is typically achieved through **abstract classes** and **interfaces**.

Example:

abstract class Animal {

abstract void sound();

}

class Dog extends Animal {

@Override

void sound() {

System.out.println("Woof");

}

}

Here, Animal class provides an abstract method sound() that must be implemented by any class that extends Animal. We don't need to know the details of sound() in the Animal class, just that every animal will have a sound.

1. **Encapsulation**
   * Encapsulation is the concept of bundling the data (attributes) and methods (functions) that manipulate the data into a single unit, known as a class.
   * It also refers to restricting access to certain components (data) of the class by using access modifiers like private, protected, and public.

Example:

class Car {

private String color; // Encapsulated attribute

public String getColor() {

return color; // Getter

}

public void setColor(String color) {

this.color = color; // Setter

}

}

public class Main {

public static void main(String[] args) {

Car car = new Car();

car.setColor("Red"); // Accessing through setter

System.out.println("Car color: " + car.getColor()); // Accessing through getter

}

}

In this example, the color attribute is private and cannot be directly accessed from outside the class. We use the getColor() and setColor() methods to interact with it.

1. **Inheritance (IS-A Relationship)**
   * **Inheritance** allows one class (the child class) to inherit the attributes and behaviors (methods) of another class (the parent class).
   * The "IS-A" relationship represents the idea that the child class is a type of the parent class.

Example:

class Animal {

void eat() {

System.out.println("This animal eats food.");

}

}

class Dog extends Animal {

void bark() {

System.out.println("The dog barks.");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.eat(); // Inherited method

dog.bark(); // Dog-specific method

}

}

Here, Dog inherits from Animal, meaning a Dog **is an** Animal. The Dog class can use the eat() method inherited from Animal while also defining its own method bark().

1. **Polymorphism**
   * **Polymorphism** means "many forms" and allows objects of different classes to be treated as objects of a common superclass.
   * It can be achieved in two ways:
     + **Method Overloading**: Same method name but different parameters.
     + **Method Overriding**: A subclass provides a specific implementation of a method already defined in its superclass.

Example (Method Overriding):

class Animal {

void speak() {

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

}

}

class Dog extends Animal {

@Override

void speak() {

System.out.println("Woof");

}

}

class Cat extends Animal {

@Override

void speak() {

System.out.println("Meow");

}

}

public class Main {

public static void main(String[] args) {

Animal myAnimal = new Animal();

Animal myDog = new Dog();

Animal myCat = new Cat();

myAnimal.speak(); // Animal makes a sound

myDog.speak(); // Woof

myCat.speak(); // Meow

}

}

Here, both Dog and Cat are overriding the speak() method from the Animal class. Even though they are treated as Animal type, they each perform their own specific version of speak(), demonstrating polymorphism.

**Containment (HAS-A Relationship)**

* + **Containment** (also known as composition) refers to the relationship where one object contains references to other objects. This represents the "HAS-A" relationship.
  + For example, a Car object may "have a" Engine object as part of it.

Example:

class Engine {

void start() {

System.out.println("Engine starts.");

}

}

class Car {

private Engine engine; // Car HAS A Engine

Car() {

engine = new Engine();

}

void drive() {

engine.start();

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

}

}

public class Main {

public static void main(String[] args) {

Car car = new Car();

car.drive();

}

}

In this example, the Car class contains an instance of the Engine class, meaning a Car **has an** Engine.