Java OOPS Concepts: ALL in One (**In Brief**)

Table of Contents

---

1. Introduction to OOPS

Object-Oriented Programming System (OOPS) is a programming paradigm designed to structure software in a modular and reusable manner. It models real-world entities and their interactions through four core principles:

Core Principles

- **Encapsulation**: Encapsulation combines data (fields) and methods (functions) into a single unit called a class. This principle hides the internal state of the object and requires all interaction to be performed through an object's methods. It improves code maintainability and enhances security by preventing external components from directly accessing or modifying an object's internal state.

    - **Advantages**:
        - **Controlled Access**: By using access modifiers (`private`, `protected`, `public`), you can control the visibility and access level of class members.
        - **Increased Flexibility**: Methods within the class can be modified without affecting external code.
        - **Improved Maintainability**: Encapsulation helps in keeping the code modular and manageable, making it easier to debug and maintain.

- **Abstraction**: Abstraction involves hiding the complex implementation details of an object and exposing only the necessary features. It allows you to focus on interactions rather than implementation specifics.

    - **Advantages**:
        - **Simplifies Code**: By providing a simplified interface to interact with complex systems.
        - **Reduces Complexity**: Helps in managing large codebases by breaking down functionality into simpler, abstracted components.

- **Inheritance**: Inheritance allows a new class (subclass) to inherit the properties and methods of an existing class (superclass). It promotes code reusability and establishes a natural hierarchy among classes.

  - **Types**:
    - **Single Inheritance**: A class inherits from a single superclass.
    - **Multilevel Inheritance**: A class inherits from another class, which itself inherits from another class.
    - **Hierarchical Inheritance**: Multiple classes inherit from a single superclass.
    - **Hybrid Inheritance**: A combination of multiple types of inheritance.

- **Polymorphism**: Polymorphism allows objects to be treated as instances of their parent class, rather than their actual class. It enables a single method to operate in multiple ways based on the object invoking it.

  - **Types**:
    - **Compile-time Polymorphism (Method Overloading)**: Multiple methods with the same name but different parameter lists.
    - **Runtime Polymorphism (Method Overriding)**: Overriding a method in a subclass to provide specific functionality.

Why OOPS Matters

- **Modularity**: OOPS enables you to develop modular code. Each class represents a separate module with specific functionality.
- **Reusability**: By using inheritance, you can reuse code across different classes, reducing redundancy.
- **Flexibility**: Polymorphism allows you to write flexible and easily extendable code.
- **Maintenance**: Encapsulation makes it easier to manage changes in code, as modifications are confined to specific classes.

---

2. Classes and Objects

In Java, a **class** is a blueprint that defines the structure and behavior of objects. **Objects** are instances of these classes and encapsulate the state and behavior defined by the class.

Class Definition

A class in Java is defined using the `class` keyword, followed by the class name and its body enclosed in curly braces.

```java
public class Person {
    // Fields (attributes)
    String name;
    int age;

    // Method (behavior)
    void displayInfo() {
        System.out.println("Name: " + name);
```

```
            System.out.println("Age: " + age);
        }
    }
```

Object Creation

Objects are created using the `new` keyword followed by the class constructor.

```
public class Main {
    public static void main(String[] args) {
        // Creating an object of the Person class
        Person person1 = new Person();
        person1.name = "Alice";
        person1.age = 30;
        person1.displayInfo();
    }
}
```

Additional Concepts

- **Constructors**: Special methods called during object creation to initialize object state. They can be overloaded to handle different initialization scenarios.
    - **Default Constructor**: Provided by Java if no constructors are defined by the user. It initializes object members to default values.
    - **Parameterized Constructor**: Allows passing arguments to initialize object fields.

    ```
    public class Car {
        String model;
        int year;

        // Default Constructor
        Car() {
            model = "Unknown";
            year = 2000;
        }

        // Parameterized Constructor
        Car(String model, int year) {
            this.model = model;
            this.year = year;
        }
    }
    ```

- `this` **Keyword**: Refers to the current instance of the class. It is used to differentiate between instance variables and method parameters.

```
public class Book {
    String title;

    void setTitle(String title) {
        this.title = title; // 'this' distinguishes between the instance
variable and the method parameter
    }
}
```

- **Memory Allocation**: In Java, objects are created on the heap memory, whereas primitive data types are stored in the stack memory.
- **Static Members**: Belong to the class rather than individual instances. Static fields and methods are shared among all instances of the class.

```
public class Counter {
    static int count = 0;

    void increment() {
        count++;
    }
}
```

---

3. Encapsulation

Encapsulation is the practice of wrapping data (fields) and methods (functions) into a single unit or class. It restricts direct access to some of an object's components and provides a controlled way to access and modify them.

Key Concepts

- **Data Hiding**: By marking fields as `private`, you prevent external classes from accessing or modifying them directly. Instead, you provide public getter and setter methods to interact with these fields.

```
public class Student {
    private String name;
    private int age;

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

    // Setter method for name
    public void setName(String name) {
        this.name = name;
    }
    // Getter method for age
    public int getAge() {
        return age;
    }

    // Setter method for age
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        }
    }
}
```

- **Access Modifiers**: Control the visibility of class members:
  - `private`: Accessible only within the same class.
  - `protected`: Accessible within the same package and subclasses.
  - `public`: Accessible from any class.
- **Mutability**: Encapsulation allows controlled modification of data through setters, ensuring data integrity and validation.

Differential Table: Encapsulation vs Abstraction

Feature	Encapsulation	Abstraction
Focus	Hides data and provides methods to manipulate it	Hides complex details and reveals only necessary features
Access Control	Controlled via access modifiers (private, public, etc.)	Focuses on exposing functionality while hiding details
Implementation	Actual code for data manipulation is present	No actual implementation (abstract methods)
Flexibility	Offers control over data access and modification	Simplifies interface by hiding internal workings
Code Example	`private int age; public void setAge(int age)`	`abstract void run();`
Purpose	Protects and manages data access	Abstracts complex functionalities

4. Inheritance

Inheritance is a mechanism where a new class inherits the properties and behaviors of an existing class. This helps in establishing a hierarchical relationship among classes and promotes code reuse.

Types of Inheritance

- **Single Inheritance**: A class inherits from a single superclass.

```
public class Animal {
    void eat() {
        System.out.println("This animal eats.");
    }
}

public class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks.");
    }
}
```

- **Multilevel Inheritance**: A class inherits from another class, which itself inherits from another class.

```
public class Animal {
    void eat() {
        System.out.println("This animal eats.");
    }
}

public class Mammal extends Animal {
    void breathe() {
        System.out.println("This mammal breathes.");
    }
}

public class Dog extends Mammal {
    void bark() {
        System.out.println("Dog barks.");
    }
}
```

- **Hierarchical Inheritance**: Multiple classes inherit from a single superclass.

```
public class Vehicle {
    void start() {
        System.out.println("Vehicle starts.");
    }
}

public class Car extends Vehicle {
    void drive() {
        System.out.println("Car drives.");
    }
}

public class Bike extends Vehicle {
    void ride() {
        System.out.println("Bike rides.");
    }
}
```

- **Hybrid Inheritance**: A combination of multiple types of inheritance. Java does not support multiple inheritance through classes to avoid complexity, but it allows multiple inheritance through interfaces.

Additional Concepts

- **Super Keyword**: Used to access members (fields and

methods) of the parent class from the child class.

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

    public class Child extends Parent {
        void display() {
            super.display(); // Calls the display method of Parent class
            System.out.println("Child class");
        }
    }
```

- **Method Overriding**: Subclass provides a specific implementation of a method that is already defined in its superclass.

```
public class Parent {
    void show() {
        System.out.println("Parent's show method.");
    }
}

public class Child extends Parent {
    @Override
    void show() {
        System.out.println("Child's show method.");
    }
}
```

- **Final Keyword**: Used to prevent further inheritance of a class or overriding of methods.

```
public final class ImmutableClass {
    // Class cannot be subclassed
}
```

```
public class Base {
    public final void finalMethod() {
        // Method cannot be overridden
    }
}
```

Differential Table: Inheritance vs Composition

Feature	Inheritance	Composition
Relationship Type	Represents an "Is-a" relationship	Represents a "Has-a" relationship
Flexibility	Less flexible, changes in parent class affect child classes	More flexible, with loosely coupled class relationships
Code Reusability	High, but can become complex with deep hierarchies	High reusability, with easier maintenance
Ease of Maintenance	Harder to maintain with complex inheritance structures	Easier to manage, reducing dependencies
Control	Inherited methods can be overridden	Better control with object composition
Example	Dog is an Animal	Car has an Engine

5. Polymorphism

Polymorphism is a concept that allows methods to do different things based on the object's data. It is implemented through method overloading and overriding.

Compile-time Polymorphism (Method Overloading)

Method overloading occurs when multiple methods in a class have the same name but different parameter lists. It is resolved during compile time.

- **Examples**:

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

    double add(double a, double b) {
        return a + b;
    }

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

Runtime Polymorphism (Method Overriding)

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. It is resolved at runtime.

- **Examples**:

```java
public class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound.");
    }
}

public class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks.");
    }
}

public class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Cat meows.");
    }
}

public class TestPolymorphism {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        myDog.makeSound(); // Outputs: Dog barks.
        myCat.makeSound(); // Outputs: Cat meows.
    }
}
```

Additional Concepts

- **Upcasting**: Converting a subclass object to its superclass type. This is safe and occurs implicitly.

```java
Animal myAnimal = new Dog(); // Upcasting
```

- **Downcasting**: Converting a superclass reference back to a subclass type. It requires explicit casting and can throw `ClassCastException` if the object is not of the subclass type.

```
Dog myDog = (Dog) myAnimal; // Downcasting
```

Differential Table: Compile-time vs Runtime Polymorphism

Feature	Compile-time Polymorphism	Runtime Polymorphism
Definition	Achieved through method overloading	Achieved through method overriding
Binding Time	Early binding (at compile-time)	Late binding (at runtime)
Flexibility	Less flexible, method signatures must be known beforehand	More flexible, depends on the actual object type at runtime
Performance	Faster, as method resolution is done at compile-time	Slower, due to method resolution during runtime
Use Case	Behavior determined at compile time	Behavior determined dynamically at runtime
Example	`add(int a, int b)` and `add(double a, double b)`	Overriding `toString()` method in subclasses

---

6. Abstraction

Abstraction involves hiding the implementation details and exposing only the necessary functionality. It can be achieved using abstract classes and interfaces.

Abstract Classes

An abstract class cannot be instantiated and can contain abstract methods (methods without implementation) and concrete methods.

- **Examples**:

```
abstract class Shape {
    abstract void draw(); // Abstract method

    void display() {
        System.out.println("Displaying shape.");
    }
}

class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing Circle.");
    }
}
```

```java
class TestAbstraction {
    public static void main(String[] args) {
        Shape myShape = new Circle();
        myShape.draw(); // Outputs: Drawing Circle.
        myShape.display(); // Outputs: Displaying shape.
    }
}
```

Interfaces

An interface is a reference type in Java that can contain only constants, method signatures, default methods, and static methods. Interfaces cannot contain instance fields or constructors.

- **Examples**:

```java
interface Drawable {
    void draw(); // Abstract method

    default void print() {
        System.out.println("Printing...");
    }
}

class Rectangle implements Drawable {
    @Override
    public void draw() {
        System.out.println("Drawing Rectangle.");
    }
}

class TestInterface {
    public static void main(String[] args) {
        Drawable myRectangle = new Rectangle();
        myRectangle.draw(); // Outputs: Drawing Rectangle.
        myRectangle.print(); // Outputs: Printing...
    }
}
```

- **Functional Interfaces**: Interfaces with exactly one abstract method, which can be implemented using lambda expressions.

```java
@FunctionalInterface
interface MathOperation {
    int operate(int a, int b);
}

public class TestFunctionalInterface {
    public static void main(String[] args) {
        MathOperation addition = (a, b) -> a + b;
        System.out.println("Sum: " + addition.operate(5, 3)); // Outputs: Sum: 8
    }
}
```

Differential Table: Abstract Class vs Interface

Feature	Abstract Class	Interface
Methods	Can have both abstract and concrete methods	Only abstract methods (default methods allowed from Java 8)
Multiple Inheritance	Not supported	Supported
Fields	Can have instance variables	Can only have constants
Access Modifiers	Methods can have any access modifier	Methods are `public` and fields are `final` by default
Constructor	Can have constructors	Cannot have constructors
Example	`abstract class Vehicle`	`interface Drawable`

---

7. Real-world Use Cases of OOPS

Understanding OOPS principles is crucial in applying design patterns and architectural principles that facilitate scalable and maintainable software development.

Design Patterns

- **Factory Pattern**: Defines an interface for creating objects, but allows subclasses to alter the type of created objects.

```
interface Product {
    void create();
}

class ConcreteProductA implements Product {
    public void create() {
        System.out.println("Product A created.");
    }
}

class ConcreteProductB implements Product {
    public void create() {
        System.out.println("Product B created.");
    }
}

class ProductFactory {
    public Product getProduct(String type) {
        if (type.equals("A")) {
            return new ConcreteProductA();
        } else if (type.equals("B")) {
            return new ConcreteProductB();
        }
        return null;
    }
}
```

- **Singleton Pattern**: Ensures a class has only one instance and provides a global point of access to it.

```
public class Singleton {
    private static Singleton instance;

    private Singleton() { }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

- **Observer Pattern**: Defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically.

```
import java.util.ArrayList;
import java.util.List;

interface Observer {
    void update(String message);
}

class ConcreteObserver implements Observer {
    private String name;

    public ConcreteObserver(String name) {
        this.name = name;
    }

    public void update(String message) {
        System.out.println(name + " received: " + message);
    }
}

class Subject {
    private List<Observer> observers = new ArrayList<>();
    void addObserver(Observer observer) {

        observers.add(observer);
    }

    void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}
```

Best Practices

- **Follow SOLID Principles**: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion.
- **Code Reviews**: Regularly review code to ensure adherence to OOPS principles.
- **Documentation**: Document classes and methods to enhance readability and maintainability.

Conclusion

Mastering OOPS concepts allows developers to design robust, scalable, and maintainable software systems. Understanding and applying these principles effectively is key to becoming a proficient Java developer.