

# Object Oriented Programming (OOP)

## Core OOP Foundations

### 1. What is OOP?

- a. It is a programming paradigm that organizes code around objects, which represent real-world or conceptual entities with both state and behavior.

### 2. Why is OOP used? What problems does it solve compared to procedural programming? How?

- a. OOP is used to manage complex and growing software systems by organizing code around objects that combine data and behavior.
- b. Compared to procedural programming, it improves maintainability and scalability by reducing tight coupling, minimizing code duplication through reuse, and making systems easier to extend and modify.
- c. Concepts like encapsulation, abstraction, inheritance, and polymorphism help model real-world entities, isolate changes, and add new features without breaking existing code, which becomes critical in large applications and team-based development.

### 3. What's a class?

- a. A blueprint/template that defines an object's properties (data) and methods (behavior). It describes what an object will have and what it can do, but it doesn't represent a real instance until an object is created from it.

### 4. What is an object?

- a. An instance of a class. It represents a real, usable entity created from the class blueprint, with actual values for its properties and the ability to use the class's methods.

### 5. What is an instance, and how is it different from a class and an object?

- a. A specific realization of a class in memory, with its own state.
- b. Class -> blueprint. Object -> actual entity created from the blueprint. Instance -> concrete, memory-allocated occurrence of the object.
- c. Object and instance usually mean the same thing, with instance emphasizing memory and state.

## Example of Class and Object

```
class Car {                                // Class (blueprint)
    void drive() {
        System.out.println("Driving");
    }
}

public class Main {
```

```
public static void main(String[] args) {
    Car myCar = new Car(); // Object / instance
    myCar.drive();
}
```

## Class Structure & Basics

### 6. What is a constructor in a class?

- a. A special method in a class that runs automatically when an object is created using the 'new' method. It is used to initialize that object's state (set initial values for variables) and has the same name as the class in Java.

### 7. Is a constructor required for a class to work?

- a. It is not required.

### 8. What happens if you don't define a constructor?

- a. If you don't define one, Java provides a default constructor that initializes the object with default values.
- b. However, if you need custom initialization, you must define your own constructor.

### 9. What is the difference between fields (instance variables) and methods?

- a. Fields: store an object's data or state, such as its properties.
- b. Methods: define the object's behavior, meaning what the object can do or how it operates on that data.

### Example of Constructor NOT explicitly defined (default constructor used)

```
class Person {
    String name;
}

class Main1 {
    public static void main(String[] args) {
        Person p = new Person(); // name = null
    }
}
```

### Example of Constructor explicitly defined (custom initialization)

```
class Person {
    String name;

    Person(String name) { // Constructor
        this.name = name;
    }
}
```

```
}
```

  

```
class Main2 {
```

```
    public static void main(String[] args) {
```

```
        Person p = new Person("Dyana"); // name initialized
```

```
    }
```

```
}
```

## Encapsulation

### 10. What is encapsulation?

- a. Bundling data and methods together and restricting direct access to internal state to protect object integrity.

### 11. How does encapsulation work in practice?

- a. Fields are kept private, and access is controlled through public methods (getters/setters or behavior methods) that enforce rules and validation.

### 12. Explain encapsulation with a real-world example.

- a. A bank account: that balance isn't directly editable. You use deposit() or withdraw() methods, which check rules (no negative balance), instead of changing the balance yourself.

### 13. What are the pros of encapsulation?

- a. Protects data from invalid states
- b. Improves maintainability and readability
- c. Reduces coupling between components
- d. Makes refactoring safer (internal changes don't break users)

### 14. What are the cons or tradeoffs of encapsulation?

- a. Slightly more boilerplate (getters/setters)
- b. Can feel restrictive or verbose for very small/simple programs
- c. Minor performance overhead in extreme cases

### 15. How do access modifiers support encapsulation?

- a. Access modifiers (private, protected, public) define who can access what, ensuring internal details are hidden while exposing only what's necessary.

## Example of Encapsulation

```
class BankAccount {
```

```
    private double balance; // encapsulated data
```

  

```
    public BankAccount(double balance) {
```

```
        this.balance = balance;
```

```
}
```

```

public void deposit(double amount) {    // controlled access
    if (amount > 0) {
        balance += amount;
    }
}

public double getBalance() {           // read-only access
    return balance;
}
}

```

## Abstraction

**16. What is abstraction?**

- a. Showing only what the user needs to know and hiding implementation details, focusing on *what* an object does rather than *how* it does it.

**17. How is abstraction different from encapsulation?**

- a. Abstraction -> hides complexity (design-level: *what* is exposed)
- b. Encapsulation -> hides data (implementation-level: *how* it's protected)

**18. What problems does abstraction solve?**

- a. Reduces complexity in large systems
- b. Allows teams to work independently
- c. Makes systems easier to extend and change
- d. Encourages clear contracts (interfaces)

**19. What are the pros and cons of abstraction?**

- a. Pros: cleaner design, better scalability, easier maintenance
- b. Cons: over-abstraction can add unnecessary layers and make code harder to trace

**20. How would you explain abstraction to someone with no programming background?**

- a. Using a car, you know how to steer, brake, and accelerate, but you don't need to understand the engine mechanics. You use what it does, not how it works.

## Example of Abstraction using Abstract Class

```

abstract class Vehicle {
    abstract void move();    // what it does
}

class Car extends Vehicle {
}

```

```

        void move() {           // how it does it
            System.out.println("Car is driving");
        }
    }

public class Main {
    public static void main(String[] args) {
        Vehicle v = new Car();    // abstraction in action
        v.move();
    }
}

```

## Abstract Classes

**21. What is an abstract class?**

- a. A class that cannot be instantiated and is meant to be extended, providing a base template for subclasses.

**22. How does an abstract class help achieve abstraction?**

- a. It defines abstract methods (what must be done) while leaving the implementation (how) to child classes.

**23. Can an abstract class have implemented methods?**

- a. Yes, it can have both abstract and concrete methods.

**24. Can an abstract class store state?**

- a. Yes, it can have instance variables (fields) just like a normal class.

**25. When would you choose an abstract class over an interface?**

- a. Abstract class -> when you need shared code, shared state, constructors, or protected members, and when subclasses have a strong “is-a” relationship and aren’t required to support multiple interfaces.

## Interfaces

**26. What is an interface?**

- a. A contract that defines what methods a class must implement, without providing implementation details.

**27. Why are interfaces used?**

- a. They define capabilities rather than inheritance, enabling flexible design and consistent behavior across unrelated classes.

**28. What are the benefits of interfaces in large systems?**

- a. Clear contracts between components
- b. Easier team collaboration and parallel development
- c. Improved testability (mocking)

- d. Safer extensibility without breaking code

**29. How do interfaces help achieve loose coupling?**

- a. Code depends on the interface, not the implementation, so implementations can be swapped without changing client code.

**30. Why does Java allow multiple interfaces but not multiple inheritance of classes?**

- a. Multiple class inheritance can cause ambiguity (diamond problem), while interfaces avoid this by not sharing state or conflicting implementations.

**31. When should you use an interface instead of an abstract class?**

- a. Use an interface when you need multiple inheritance, no shared data, or want to define a pure contract across unrelated classes.

**32. What are the downsides of interfaces?**

- a. No instance state
- b. More boilerplate in small projects
- c. Changes to interfaces require updates to all implementers

### Example of Interfaces

```
interface Payment {  
    void pay();  
}  
  
class CreditCardPayment implements Payment {  
    public void pay() {  
        System.out.println("Paid with credit card");  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Payment p = new CreditCardPayment(); // interface reference  
        p.pay();  
    }  
}
```

## Inheritance

**33. What is inheritance?**

- a. A mechanism where a subclass acquires fields and methods of another class, allowing code reuse and hierarchical relationships

**34. Explain inheritance with an example.**

- a. A Car class inheriting from Vehicle - the car is a vehicle and can resume common behavior like move()

**35. What is the difference between superclass, subclass, and base class?**

- a. Superclass / base class -> the parent class being inherited from
- b. Subclass -> the child class that extends it

**36. What does an “is-a” relationship mean?**

- a. It means the subclass can be substituted wherever the superclass is expected  
(Car is-a Vehicle)

**37. What are the benefits of inheritance?**

- a. Code reuse
- b. Clear hierarchy and shared behavior
- c. Enables polymorphism

**38. What are the drawbacks of inheritance?**

- a. Tight coupling between parent and child
- b. Harder to change without side effects
- c. Can encourage deep, rigid hierarchies

**39. What is the fragile base class problem?**

- a. Changes in a base class can unexpectedly break subclasses, even if their code didn't change

**40. When does inheritance become a bad design choice?**

- a. When relationships are not truly is-a, behavior varies widely, or flexibility is needed - composition is often better in these cases.

### Example of Inheritance

```
class Vehicle {  
    void move() {  
        System.out.println("Moving");  
    }  
}  
  
class Car extends Vehicle {  
    void honk() {  
        System.out.println("Honk!");  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Car c = new Car(); // inherits move()  
        c.move();  
        c.honk();  
    }  
}
```

## Composition vs Inheritance

### 41. What is composition?

- a. A design principle where a class contains other objects and delegates behavior to them instead of inheriting from them.

### 42. What is the difference between composition and inheritance?

- a. Composition -> has-a relationship (uses another object)
- b. Inheritance -> is-a relationship (extends another class)

### 43. What does a “has-a” relationship mean?

- a. It means an object owns or uses another object as part of its functionality

### 44. Why is composition often preferred over inheritance?

- a. Looser coupling
- b. More flexible and easier to change
- c. Avoids fragile base class problems
- d. Encourages reuse without rigid hierarchies

### 45. Give an example where composition is a better choice than inheritance.

- a. A **Car** shouldn't inherit from **Engine**. Instead, it **has an Engine** and delegates starting and stopping behavior to it - composition fits naturally here and allows engines to be swapped without changing the car's inheritance structure.

## Example of Composition

```
class Engine {  
    void start() {  
        System.out.println("Engine started");  
    }  
}  
  
class Car {  
    private Engine engine; // Car HAS an Engine (composition)  
  
    Car() {  
        engine = new Engine();  
    }  
  
    void drive() {  
        engine.start(); // delegate behavior  
        System.out.println("Car is driving");  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Car car = new Car();  
        car.drive();  
    }  
}
```

## Polymorphism

### 46. What is polymorphism?

- a. The same method call can behave differently depending on the object - one interface, many implementations

### 47. How does polymorphism work in OOP?

- a. A parent type reference (interface or superclass) points to different child objects, and the correct method implementation is chosen automatically

### 48. Explain polymorphism with a real-world example.

- a. A remote control: the same “power” button turns on a TV, AC, or speaker - same action, different behavior.

### 49. How would you explain polymorphism to someone who doesn't know OOP?

- a. It's like saying “drive” - a car drives on roads, a boat “drives” on water, a plane “drives” in the air, but the command is the same.

### 50. What is runtime polymorphism?

- a. Occurs when method calls are resolved at runtime using method **overriding** (dynamic dispatch)

### 51. What is compile-time polymorphism?

- a. Occurs at compile time, usually through method **overloading**, where the method chosen depends on parameter type or counts.

## Example of Polymorphism using Override (runtime polymorphism)

```
class Animal {  
    void sound() {  
        System.out.println("Animal makes a sound");  
    }  
}  
  
class Dog extends Animal {  
    @Override  
    void sound() {  
        System.out.println("Dog barks");  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Animal a = new Dog(); // polymorphism  
        a.sound(); // calls Dog's version at runtime  
    }  
}
```

## Overloading vs Overriding

### 52. What is method overloading?

- a. Defining multiple methods with the same name in the same class but with different parameter lists (number, type, or order). It's resolved at compile time.

### 53. What is method overriding?

- a. When a subclass provides its own implementation of a method already defined in its superclass, using the same method signature. It's resolved at runtime.

### 54. What is the difference between overloading and overriding?

- a. Overloading -> same class, different parameters, compile-time
- b. Overriding -> subclass, same signature, runtime

### 55. How does overloading affect classes?

- a. It improves usability and readability by allowing the same logical operation to handle different inputs without creating new method names

### 56. What rules must be followed when overriding a method?

- a. Method signature must match exactly
- b. Return type must be the same or covariant
- c. Access level cannot be more restrictive
- d. Cannot override final methods
- e. Overridden method can't throw broader checked exceptions

### 57. Can static methods be overridden?

- a. No. static methods are hidden, not overridden, because they belong to the class, not instances.

### 58. How do access modifiers affect overriding?

- a. You can increase visibility (e.g. protected -> public) but not decrease it.

### 59. What happens if method signatures don't match?

- a. It becomes overloading, not overriding, and polymorphism will not occur.

## Example of Overloading

```
class Calculator {
```

```

// Method overloading: same name, different parameters
int add(int a, int b) {
    return a + b;
}

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

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

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(2, 3));           // calls add(int, int)
        System.out.println(calc.add(1, 2, 3));         // calls add(int, int, int)
        System.out.println(calc.add(2.5, 3.5));       // calls add(double, double)
    }
}

```

## Interfaces vs Abstract Classes

**60. What is the difference between an abstract class and an interface?**

- Abstract class -> can have state, constructors, and implemented methods; represents a strong is-a relationship
- Interface -> defines a pure contract (no instance state), focuses on what a class can do

**61. When would you use an abstract class?**

- When classes share common behavior or fields, need constructors or protected methods, and have a close conceptual relationship

**62. When would you use an interface?**

- To define capabilities across unrelated classes, support multiple inheritance, enable loose coupling, or when you want a stable API contract

**63. Why can a class implement multiple interfaces but only extend one class?**

- a. Multiple class inheritance can cause ambiguity and state conflicts (diamond problem). Interfaces avoid this because they don't carry instance state, making multiple inheritance safe.

## Dependency Management & Design Principles

### 64. What is tight coupling vs loose coupling?

- a. Tight coupling -> classes depend on concrete implementations, making changes risky
- b. Loose coupling -> classes depend on abstractions (interfaces), making systems flexible

### 65. What is Dependency Injection (DI)?

- a.

### 66. What is Inversion of Control (IoC)?

- a.

### 67. How are DI and IoC related?

- a.

### 68. Why does dependency injection improve testability?

- a.

### 69. Give a simple DI example without using a framework.

- a.