CodInClub

Powered By BridgeLabz

Introduction of Inheritance

Assisted Problems

1. **Animal Hierarchy**
   - **Description**: Create a hierarchy where `Animal` is the superclass, and `Dog`, `Cat`, and `Bird` are subclasses. Each subclass has a unique behavior.
   - **Tasks**:
     - Define a superclass `Animal` with attributes `name` and `age`, and a method `makeSound()`.
     - Define subclasses `Dog`, `Cat`, and `Bird`, each with a unique implementation of `makeSound()`.
   - **Goal**: Learn basic inheritance, method overriding, and polymorphism with simple classes.

```
public class Animal {
 String name;
 int age;
public Animal(String name,int age){
    this.name = name;
    this.age = age;
}
void makeSound(){
    System.out.println("Each animal produces a specific sound.");
}
void sounds(){
    System.out.println("hello");
}
}
```

```
public class Bird extends Animal{
    Bird(String name, int age){
        super(name, age);
    }
    void makeSound(){
        System.out.println(name+" chirps.(age= "+age+")");
    }
}
```

```java
public class Cat extends Animal{
    Cat(String name, int age){
        super(name, age);
    }
    void makeSound(){
        System.out.println(name+" sounds like meow.(age= "+age+")");
    }
}
```

```java
public class Dog extends Animal{
    Dog(String name, int age){
        super(name, age);
    }
    void makeSound(){
        System.out.println(name+" barks.(age= "+age+")");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        System.out.println(" ");
        // Animal a =new Animal(null, 0);
        Dog d = new Dog("Muffy", 5);
        d.sounds();
        d.makeSound();
        Cat c= new Cat("Billu", 4);
        c.makeSound();
        Bird b= new Bird("Tweety",3);
        b.makeSound();
    }
}
```

2. **Employee Management System**
   - **Description**: Create an Employee hierarchy for different employee types such as Manager, Developer, and Intern.
   - **Tasks**:
     - Define a base class Employee with attributes like name, id, and salary, and a method displayDetails().
     - Define subclasses Manager, Developer, and Intern with unique attributes for each, like teamSize for Manager and programmingLanguage for Developer.
   - **Goal**: Practice inheritance by creating subclasses with specific attributes and overriding superclass methods.

```
public class Employee {
    String name;
    int id;
    int salary;
     Employee(String name, int id, int salary){
        this.name = name;
        this.id =id;
        this.salary = salary;
    }
     void displayDetails(){
        System.out.println(" name= "+name+" id= "+id+" salary= "+salary);
    }
}
```

```
public class Manager extends Employee{
 int teamSize;
 Manager(String name, int id, int salary){
    super(name, id, salary);
 }
 @Override
 void displayDetails(){
    super.displayDetails();
    System.out.println("team Size= "+teamSize);
 }
}
```

```
public class Developer extends Employee {
    String programmingLanguage ;
    Developer(String name, int id, int salary, String programmingLanguage){
        super(name,id,salary);
        this.programmingLanguage = programmingLanguage;
    }
    @Override
     void displayDetails(){
        super.displayDetails();
        System.out.println("programming language= "+programmingLanguage);
    }
}
```

```
public class Intern extends Employee{
    Intern(String name, int id, int salary){
        super(name,id,salary);
    }
    @Override
    void displayDetails(){
        super.displayDetails();
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        System.out.println(" ");
        Developer d = new Developer("Prapat", 1001, 1500000, "java");
        d.displayDetails();
        Manager m = new Manager("amit", 5005, 4560000);
        m.displayDetails();
        Intern i =new Intern("Vinita", 3003, 120000);
        i.displayDetails();
    }
}
```

3. **Vehicle and Transport System**
   - **Description**: Design a vehicle hierarchy where `Vehicle` is the superclass, and `Car`, `Truck`, and `Motorcycle` are subclasses with unique attributes.
   - **Tasks**:
     - Define a superclass `Vehicle` with `maxSpeed` and `fuelType` attributes and a method `displayInfo()`.
     - Define subclasses `Car`, `Truck`, and `Motorcycle`, each with additional attributes, such as `seatCapacity` for `Car`.
     - Demonstrate polymorphism by storing objects of different subclasses in an array of `Vehicle` type and calling `displayInfo()` on each.
   - **Goal**: Understand how inheritance helps in organizing shared and unique features across subclasses and use polymorphism for dynamic method calls.

```
public class Vechicle {
    int maxSpeed;
    String fuelType;
    Vechicle(int maxSpeed, String fuelType){
        this.maxSpeed = maxSpeed;
        this.fuelType=fuelType;
    }
    void displayInfo(){
        System.out.println("Max Speed= "+maxSpeed+" fuel type= "+fuelType);
    }
}
```

```
public class Truck extends Vechicle {
    int loadingCapacity;
    Truck(int maxSpeed, String fuelType, int loadingCapacity){
        super(maxSpeed,fuelType);
        this.loadingCapacity = loadingCapacity;
    }
    @Override
    void displayInfo(){
        super.displayInfo();
        System.out.println("loading capacity= "+loadingCapacity);
    }
}
```

```
public class MotorCycle extends Vechicle{
    int engineCC;
    MotorCycle(int maxSpeed, String fuelType, int engineCC){
        super(maxSpeed, fuelType);
        this.engineCC=engineCC;
    }
    @Override
    void displayInfo(){
        super.displayInfo();
        System.out.println("engine capacity= "+engineCC);
    }
}
```

```
public class Car extends Vechicle{
    int seatCapacity;
    Car(int maxSpeed, String fuelType, int seatCapacity){
        super(maxSpeed,fuelType);
        this.seatCapacity = seatCapacity;
    }
    @Override
    void displayInfo(){
        super.displayInfo();
        System.out.println("Seat capacity= "+seatCapacity);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        System.out.println(" ");
        Car c= new Car(300,"Petrol",4);
        c.displayInfo();
        Truck t = new Truck(100, "diesel", 1000);
        t.displayInfo();
        MotorCycle mt = new MotorCycle(180, "petrol", 130);
        mt.displayInfo();
    }
}
```

Single Inheritance

Sample Problem 1: Library Management with Books and Authors

- **Description**: Model a `Book` system where `Book` is the superclass, and `Author` is a subclass.
- **Tasks**:
  - Define a superclass `Book` with attributes like `title` and `publicationYear`.
  - Define a subclass `Author` with additional attributes like `name` and `bio`.
  - Create a method `displayInfo()` to show details of the book and its author.
- **Goal**: Practice single inheritance by extending the base class and adding more specific details in the subclass.

```
public class Book {
    String title;
    int publicationYear;
    Book(String title, int publicationYear){
        this.title =title;
        this.publicationYear = publicationYear;
    }
    void displayInfo(){
        System.out.println("title of the Book= "+title);
        System.out.println("Publication Year= "+publicationYear);
    }
}
```

```
public class Author extends Book{
    String name;
    String bio;
    Author(String title, int publicationYear, String name, String bio){
        super(title, publicationYear);
        this.name = name;
        this.bio = bio;
    }
    void displayInfo(){
        super.displayInfo();
        System.out.println("name of the author= "+name);
        System.out.println("Description= "+bio);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Author a = new Author("Physics", 2009, "Selina", "Great book for JEE");
        a.displayInfo();
    }
}
```

Sample Problem 2: Smart Home Devices

- **Description**: Create a hierarchy for a smart home system where `Device` is the superclass and `Thermostat` is a subclass.
- **Tasks**:
    - Define a superclass `Device` with attributes like `deviceId` and `status`.

        - Create a subclass `Thermostat` with additional attributes like `temperatureSetting`.
        - Implement a method `displayStatus()` to show each device's current settings.
    - **Goal**: Understand single inheritance by adding specific attributes to a subclass, keeping the superclass general.

```java
public class Device {
    int deviceId ;
    String status;
    Device(int deviceId, String status){
        this.deviceId = deviceId;
        this.status = status;
    }
}
```

```java
public class Thermostat  extends Device{
    String temperatureSetting;
    Thermostat(int deviceId, String status, String temperatureSetting){
        super(deviceId, status);
        this.temperatureSetting= temperatureSetting;
    }
    void displayStatus(){
        System.out.println("Device Id:" +deviceId +" ,with status: "+ status+"
,have current temp setting: "+temperatureSetting );
    }
}
```

```java

public class Main {
    public static void main(String[] args) {
        System.out.println(" ");
        Thermostat th = new Thermostat(1001, "Room temp", "20 deg.");
        th.displayStatus();
    }
}
```

Multilevel Inheritance

Sample Problem 1: Online Retail Order Management

- **Description**: Create a multilevel hierarchy to manage orders, where `Order` is the base class, `ShippedOrder` is a subclass, and `DeliveredOrder` extends `ShippedOrder`.
- **Tasks**:
  - Define a base class `Order` with common attributes like `orderId` and `orderDate`.
  - Create a subclass `ShippedOrder` with additional attributes like `trackingNumber`.
  - Create another subclass `DeliveredOrder` extending `ShippedOrder`, adding a `deliveryDate` attribute.
  - Implement a method `getOrderStatus()` to return the current order status based on the class level.
- **Goal**: Explore multilevel inheritance, showing how attributes and methods can be added across a chain of classes.

```java
public class Order {
    int orderId;
    String orderDate;
    Order(int orderId, String orderDate){
        this.orderId =orderId;
        this.orderDate = orderDate;
    }
    void getOrderStatus(){
        System.out.println("Order placed.");
        System.out.println("OrderId= "+orderId+",order date= "+orderDate);
    }
}
```

```java
public class ShippedOrder extends Order {
    int trackingNumber;
    ShippedOrder(int orderId, String orderDate, int trackingNumber){
        super(orderId, orderDate);
        this.trackingNumber = trackingNumber;
    }
    void getOrderStatus(){
        System.out.println("order shipped. Tranking number: "+trackingNumber);
    }
}
```

```
public class DeliveredOrder extends ShippedOrder{
    String deliveryDate ;
    DeliveredOrder(int orderId, String orderDate,int trackingNumber,String
deliveryDate){
        super(orderId,orderDate, trackingNumber);
        this.deliveryDate = deliveryDate;
    }
    void getOrderStatus(){
        System.out.println("order delivered on "+deliveryDate);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        System.out.println(" ");
        Order order = new Order(1001, "2025-02-01");
        ShippedOrder shippedOrder = new ShippedOrder(1002, "2025-02-02",
451616);
        DeliveredOrder deliveredOrder = new DeliveredOrder(1003, "2025-02-03",
816864, "2025-02-05");
        order.getOrderStatus();
        shippedOrder.getOrderStatus();
        deliveredOrder.getOrderStatus();
    }
}
```

CodInClub

Powered By BridgeLabz

Sample Problem 2: Educational Course Hierarchy

- **Description**: Model a course system where `Course` is the base class, `OnlineCourse` is a subclass, and `PaidOnlineCourse` extends `OnlineCourse`.
- **Tasks**:
  - Define a superclass `Course` with attributes like `courseName` and `duration`.
  - Define `OnlineCourse` to add attributes such as `platform` and `isRecorded`.
  - Define `PaidOnlineCourse` to add `fee` and `discount`.
- **Goal**: Demonstrate how each level of inheritance builds on the previous, adding complexity to the system.

```
public class Course {
    String courseName ;
    double duration;
    Course(String courseName, double duration){
        this.courseName = courseName;
        this.duration = duration;
    }
    void display(){
        System.out.println("Course details:\n course name=
"+courseName+"\nduration= "+duration);
    }
}
```

```
public class OnlineCourse extends Course {
    String platform;
    boolean isRecorded;
    OnlineCourse(String courseName, double duration, String platform, boolean
isRecorded){
        super(courseName, duration);
        this.platform = platform;
        this.isRecorded=isRecorded;
    }
    @Override
    void display(){
        super.display();
        System.out.println("\nPlatform= "+platform+"\n Recorded or Not=
"+isRecorded);
    }
}
```

```
public class PaidOnlineCourse extends OnlineCourse {
    int fee;
    double discount;
    PaidOnlineCourse(String courseName, double duration, String platform,
boolean isRecorded, int fee, double discount){
        super(courseName, duration, platform, isRecorded);
        this.fee= fee;
        this.discount=discount;
    }
    @Override
    void display(){
        super.display();
        System.out.println("\nCourse Fee: "+fee+"\n Special discount: "+
discount);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        System.out.println(" ");
        Course c= new Course("Biology", 3.5);
        c.display();
        OnlineCourse oc = new OnlineCourse("Physics", 2.5, "Coursera", true);
        oc.display();
        PaidOnlineCourse pd = new PaidOnlineCourse("Math", 4.5, "Udemy", true,
1000, 100);
        pd.display();
    }
}
```

Hierarchical Inheritance

Sample Problem 1: Bank Account Types

- **Description**: Model a banking system with different account types using hierarchical inheritance. `BankAccount` is the superclass, with `SavingsAccount`, `CheckingAccount`, and `FixedDepositAccount` as subclasses.
- **Tasks**:
  - Define a base class `BankAccount` with attributes like `accountNumber` and `balance`.
  - Define subclasses `SavingsAccount`, `CheckingAccount`, and `FixedDepositAccount`, each with unique attributes like `interestRate` for `SavingsAccount` and `withdrawalLimit` for `CheckingAccount`.
  - Implement a method `displayAccountType()` in each subclass to specify the account type.
- **Goal**: Explore hierarchical inheritance, demonstrating how each subclass can have unique attributes while inheriting from a shared superclass.

```
class BankAccount {
    protected String accountNumber;
    protected double balance;

    public BankAccount(String accountNumber, double balance) {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }

    public void deposit(double amount) {
        balance += amount;
        System.out.println("Deposited: $" + amount + ". New Balance: $" +
balance);
```

```
    }

    public void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
            System.out.println("Withdrawn: $" + amount + ". New Balance: $" +
balance);
        } else {
            System.out.println("Insufficient balance.");
        }
    }

    public void displayBalance() {
        System.out.println("Account Number: " + accountNumber + ", Balance: $"
+ balance);
    }

    public void displayAccountType() {
        System.out.println("Generic Bank Account");
    }
}
```

```
class CheckingAccount extends BankAccount {
    private double withdrawalLimit;

    public CheckingAccount(String accountNumber, double balance, double
withdrawalLimit) {
        super(accountNumber, balance);
        this.withdrawalLimit = withdrawalLimit;
    }

    @Override
    public void withdraw(double amount) {
        if (amount > withdrawalLimit) {
            System.out.println("Withdrawal failed. Amount exceeds the limit of
$" + withdrawalLimit);
        } else {
            super.withdraw(amount);
        }
    }

    @Override
    public void displayAccountType() {
        System.out.println("This is a Checking Account.");
    }
}
```

```
class FixedDepositAccount extends BankAccount {
    private int lockInPeriod; // in months

    public FixedDepositAccount(String accountNumber, double balance, int
lockInPeriod) {
        super(accountNumber, balance);
        this.lockInPeriod = lockInPeriod;
    }

    public void withdraw(double amount) {
        System.out.println("Withdrawals are not allowed in a Fixed Deposit
Account until the lock-in period of " + lockInPeriod + " months is
completed.");
    }

    public void displayAccountType() {
        System.out.println("This is a Fixed Deposit Account.");
    }
}
```

```
class SavingsAccount extends BankAccount {
    private double interestRate;

    public SavingsAccount(String accountNumber, double balance, double
interestRate) {
        super(accountNumber, balance);
        this.interestRate = interestRate;
    }

    public void addInterest() {
        double interest = balance * (interestRate / 100);
        balance += interest;
        System.out.println("Interest added: $" + interest + ". New Balance: $"
+ balance);
    }

    public void displayAccountType() {
        System.out.println("This is a Savings Account.");
    }
}
```

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

        SavingsAccount savings = new SavingsAccount("SA12345", 1000, 5);
        CheckingAccount checking = new CheckingAccount("CA54321", 2000, 500);
        FixedDepositAccount fixedDeposit = new FixedDepositAccount("FD67890",
5000, 12);


        savings.displayAccountType();
        checking.displayAccountType();
        fixedDeposit.displayAccountType();


        System.out.println("\n--- Transactions ---");
        savings.deposit(500);
        savings.addInterest();
        savings.withdraw(200);

        System.out.println();
        checking.withdraw(600);
        checking.withdraw(400);

        System.out.println();
        fixedDeposit.withdraw(1000);
    }
}
```

Sample Problem 2: School System with Different Roles

- **Description**: Create a hierarchy for a school system where `Person` is the superclass, and `Teacher`, `Student`, and `Staff` are subclasses.
- **Tasks**:
  - Define a superclass `Person` with common attributes like `name` and `age`.
  - Define subclasses `Teacher`, `Student`, and `Staff` with specific attributes (e.g., `subject` for `Teacher` and `grade` for `Student`).
  - Each subclass should have a method like `displayRole()` that describes the role.
- **Goal**: Demonstrate hierarchical inheritance by modeling different roles in a school, each with shared and unique characteristics.

```java
class Person {
    protected String name;
    protected int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void displayDetails() {
        System.out.println("Name: " + name + ", Age: " + age);
    }

    public void displayRole() {
        System.out.println("Generic Person");
    }
}
```

```
class Staff extends Person {
    private String department;

    public Staff(String name, int age, String department) {
        super(name, age);
        this.department = department;
    }

    public void displayRole() {
        System.out.println(name + " is a Staff member in " + department + "
department");
    }
}
```

```
class Student extends Person {
    private int grade;

    public Student(String name, int age, int grade) {
        super(name, age);
        this.grade = grade;
    }

    public void displayRole() {
        System.out.println(name + " is a Student in Grade " + grade);
    }
}
```

```
class Teacher extends Person {
    private String subject;

    public Teacher(String name, int age, String subject) {
        super(name, age);
        this.subject = subject;
    }

    public void displayRole() {
        System.out.println(name + " is a Teacher, teaching " + subject);
    }
}
```

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

        Teacher teacher = new Teacher("Mr. Smith", 40, "Mathematics");
        Student student = new Student("Alice", 14, 9);
        Staff staff = new Staff("Mrs. Johnson", 35, "Administration");

        teacher.displayDetails();
        teacher.displayRole();

        System.out.println();

        student.displayDetails();
        student.displayRole();

        System.out.println();

        staff.displayDetails();
        staff.displayRole();
    }
}
```

Hybrid Inheritance (Simulating Multiple Inheritance)

Since Java doesn't support multiple inheritance directly, hybrid inheritance is typically achieved through **interfaces**.

Sample Problem 1: Restaurant Management System with Hybrid Inheritance

- **Description**: Model a restaurant system where `Person` is the superclass and `Chef` and `Waiter` are subclasses. Both `Chef` and `Waiter` should implement a `Worker` interface that requires a `performDuties()` method.

- **Tasks**:
  - Define a superclass `Person` with attributes like `name` and `id`.
  - Create an interface `Worker` with a method `performDuties()`.
  - Define subclasses `Chef` and `Waiter` that inherit from `Person` and implement the `Worker` interface, each providing a unique implementation of `performDuties()`.
- **Goal**: Practice hybrid inheritance by combining inheritance and interfaces, giving multiple behaviors to the same objects.

```java
class Person {
    protected String name;
    protected int id;

    public Person(String name, int id) {
        this.name = name;
        this.id = id;
    }

    public void displayDetails() {
        System.out.println("ID: " + id + ", Name: " + name);
    }
}
```

```java
interface Worker {
    void performDuties();
}
```

```java
class Chef extends Person implements Worker {
    private String specialty;

    public Chef(String name, int id, String specialty) {
        super(name, id);
        this.specialty = specialty;
    }

    public void performDuties() {
        System.out.println(name + " (Chef) is cooking " + specialty + " 
dishes.");
    }
}
```

```
class Waiter extends Person implements Worker {
    private int tablesAssigned;

    public Waiter(String name, int id, int tablesAssigned) {
        super(name, id);
        this.tablesAssigned = tablesAssigned;
    }

    public void performDuties() {
        System.out.println(name + " (Waiter) is serving customers at " + 
tablesAssigned + " tables.");
    }
}
```

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

        Chef chef = new Chef("Gordon Ramsay", 101, "Italian Cuisine");
        Waiter waiter = new Waiter("John Doe", 202, 5);


        chef.displayDetails();
        chef.performDuties();

        System.out.println();

        waiter.displayDetails();
        waiter.performDuties();
    }
}
```

Sample Problem 2: Vehicle Management System with Hybrid Inheritance

- **Description**: Model a vehicle system where `Vehicle` is the superclass and `ElectricVehicle` and `PetrolVehicle` are subclasses. Additionally, create a `Refuelable` interface implemented by `PetrolVehicle`.
- **Tasks**:
    - Define a superclass `Vehicle` with attributes like `maxSpeed` and `model`.
    - Create an interface `Refuelable` with a method `refuel()`.
    - Define subclasses `ElectricVehicle` and `PetrolVehicle`. `PetrolVehicle` should implement `Refuelable`, while `ElectricVehicle` include a charge`()` method.
- **Goal**: Use hybrid inheritance by having `PetrolVehicle` implement both `Vehicle` and `Refuelable`, demonstrating how Java interfaces allow adding multiple behaviors.

```
class Vehicle {
    protected String model;
    protected int maxSpeed;

    public Vehicle(String model, int maxSpeed) {
        this.model = model;
        this.maxSpeed = maxSpeed;
    }

    public void displayDetails() {
        System.out.println("Model: " + model + ", Max Speed: " + maxSpeed + "
km/h");
    }
}
```

```
interface Refuelable {
    void refuel();
}
```

```java
class PetrolVehicle extends Vehicle implements Refuelable {
    private int fuelTankCapacity;

    public PetrolVehicle(String model, int maxSpeed, int fuelTankCapacity) {
        super(model, maxSpeed);
        this.fuelTankCapacity = fuelTankCapacity;
    }

    public void refuel() {
        System.out.println(model + " (Petrol) is refueling. Tank capacity: " +
fuelTankCapacity + " liters.");
    }
}
```

```java
class ElectricVehicle extends Vehicle {
    private int batteryCapacity;

    public ElectricVehicle(String model, int maxSpeed, int batteryCapacity) {
        super(model, maxSpeed);
        this.batteryCapacity = batteryCapacity;
    }

    public void charge() {
        System.out.println(model + " (Electric) is charging. Battery capacity:
" + batteryCapacity + " kWh.");
    }
}
```

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

        ElectricVehicle tesla = new ElectricVehicle("Tesla Model S", 250, 100);
        PetrolVehicle bmw = new PetrolVehicle("BMW M3", 280, 60);

        tesla.displayDetails();
        tesla.charge();

        System.out.println();
        bmw.displayDetails();
        bmw.refuel();
    }
}
```

1. Favor Composition Over Inheritance

- Use composition instead
- instead of inheritance when a class can be described as "has-a" rather than "is-a".
- This avoids tight coupling and provides greater flexibility.

---

2. Ensure Proper Use of `is-a` Relationship

- Use inheritance only when the subclass truly extends the behavior of the superclass, maintaining the "is-a" relationship.
- Avoid misusing inheritance for code reuse.

---

3. Follow Liskov Substitution Principle

- Subclasses should be substitutable for their superclasses without breaking the application.
- Ensure overridden methods maintain the expected behavior of the superclass.

---

4. Avoid Deep Inheritance Hierarchies

- Keep the inheritance hierarchy shallow to reduce complexity and improve maintainability.
- Deep hierarchies can make debugging and understanding the code difficult.

---

5. Mark Superclass Methods `final` If Needed

- Prevent subclasses from overriding critical methods by marking them `final`.

- This ensures essential functionality remains unchanged.

---

6. Use `@Override` Annotation

- Always use `@Override` to explicitly indicate that a method is being overridden.
- This helps catch errors during compilation if the method signature is incorrect.

---

7. Minimize Public Fields in Superclasses

- Use private or protected fields with proper getters and setters.
- This prevents unintended access or modification by subclasses.

---

8. Avoid Overloading Alongside Overriding

- Overloading methods with similar names and parameters in subclasses can lead to confusion.
- Ensure clarity by distinctly separating overridden methods from overloaded ones.

---

9. Prefer Abstract Classes for Partial Implementation

- Use abstract classes to define a blueprint with partial implementation for related classes.
- Abstract classes provide flexibility while enforcing a consistent structure.

---

10. Use Interfaces for Multiple Inheritance

- Java does not support multiple inheritance through classes. Use interfaces to achieve multiple inheritance-like behavior.
- This helps avoid the "diamond problem."

---

11. Document Inheritance Behavior

- Clearly document the purpose and expected behavior of the superclass and its methods.

- Provide details on how subclasses should override or extend the methods.

---

12. Avoid Overriding Methods Unnecessarily

- Override methods only when necessary and when the subclass needs to modify or extend the behavior of the superclass.

---

13. Be Cautious with Constructors

- Call the superclass constructor explicitly in the subclass constructor using `super()`.
- Avoid calling non-final methods from constructors to prevent issues with uninitialized state in subclasses.

---

14. Use Polymorphism Effectively

- Design systems to leverage polymorphism where superclass references are used to interact with subclass objects.
- This promotes flexibility and extensibility.

---

15. Beware of Fragile Base Class Problem

- Changes to the superclass can inadvertently affect all subclasses.
- Minimize dependencies and changes to the superclass once it is widely used.

---

16. Test Subclass and Superclass Interactions

- Thoroughly test how the subclass interacts with inherited methods and state.
- Ensure changes in the subclass do not break the expected behavior of the superclass.

---

17. Avoid Inheriting from Concrete Classes

- Prefer inheriting from abstract classes or interfaces rather than concrete classes.

- This avoids tight coupling to a specific implementation.

---

18. Consider Using Delegation for Special Cases

- When specific behavior is needed in some instances but not others, delegation may be a better choice than inheritance.
- This promotes better separation of concerns.