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.

```
class Animal {
    String name;
    int age;

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

    void makeSound() {
        System.out.println("Some generic animal sound");
    }
}


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

```
}

class Cat extends Animal {
    Cat(String name, int age) {
        super(name, age);
    }
    void makeSound() {
        System.out.println(name + " meows: Meow Meow!");
    }
}

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


public class AnimalHierarchy {
    public static void main(String[] args) {
        Animal myDog = new Dog("Buddy", 3);
        Animal myCat = new Cat("Whiskers", 2);
        Animal myBird = new Bird("Tweety", 1);

        myDog.makeSound();
        myCat.makeSound();
        myBird.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.

```
class Employee {
    String name;
    int id;
    double salary;

    Employee(String name, int id, double salary) {
        this.name = name;
        this.id = id;
        this.salary = salary;
    }

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

// Subclasses
class Manager extends Employee {
    int teamSize;

    Manager(String name, int id, double salary, int teamSize) {
        super(name, id, salary);
        this.teamSize = teamSize;
    }

    void displayDetails() {
        super.displayDetails();
        System.out.println("Team Size: " + teamSize);
    }
}

class Developer extends Employee {
```

```
    String programmingLanguage;

    Developer(String name, int id, double salary, String
programmingLanguage) {
        super(name, id, salary);
        this.programmingLanguage = programmingLanguage;
    }

    void displayDetails() {
        super.displayDetails();
        System.out.println("Programming Language: " + programmingLanguage);
    }
}

class Intern extends Employee {
    String duration;

    Intern(String name, int id, double salary, String duration) {
        super(name, id, salary);
        this.duration = duration;
    }

    void displayDetails() {
        super.displayDetails();
        System.out.println("Internship Duration: " + duration);
    }
}


public class EmployeeManagement {
    public static void main(String[] args) {
        Employee emp1 = new Manager("Alice", 101, 90000, 5);
        Employee emp2 = new Developer("Bob", 102, 70000, "Java");
        Employee emp3 = new Intern("Charlie", 103, 20000, "6 months");

        emp1.displayDetails();
        emp2.displayDetails();
        emp3.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.

```
class Vehicle {
    int maxSpeed;
    String fuelType;

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

    void displayInfo() {
        System.out.println("Max Speed: " + maxSpeed + " km/h, Fuel Type: "
+ fuelType);
    }
}

class Car extends Vehicle {
    int seatCapacity;

    Car(int maxSpeed, String fuelType, int seatCapacity) {
        super(maxSpeed, fuelType);
        this.seatCapacity = seatCapacity;
    }
```

```
    void displayInfo() {
        System.out.println("Car - Max Speed: " + maxSpeed + " km/h, Fuel
Type: " + fuelType + ", Seat Capacity: " + seatCapacity);
    }
}

class Truck extends Vehicle {
    int loadCapacity;

    Truck(int maxSpeed, String fuelType, int loadCapacity) {
        super(maxSpeed, fuelType);
        this.loadCapacity = loadCapacity;
    }

    void displayInfo() {
        System.out.println("Truck - Max Speed: " + maxSpeed + " km/h, Fuel
Type: " + fuelType + ", Load Capacity: " + loadCapacity + " tons");
    }
}

class Motorcycle extends Vehicle {
    boolean hasSidecar;

    Motorcycle(int maxSpeed, String fuelType, boolean hasSidecar) {
        super(maxSpeed, fuelType);
        this.hasSidecar = hasSidecar;
    }

    void displayInfo() {
        System.out.println("Motorcycle - Max Speed: " + maxSpeed + " km/h,
Fuel Type: " + fuelType + ", Sidecar: " + (hasSidecar ? "Yes" : "No"));
    }
}

public class VehicleTransportSystem {
    public static void main(String[] args) {
        Vehicle[] vehicles = new Vehicle[3];
        vehicles[0] = new Car(180, "Petrol", 5);
        vehicles[1] = new Truck(120, "Diesel", 10);
        vehicles[2] = new Motorcycle(150, "Petrol", false);
```

```
        for (int i = 0; i < vehicles.length; i++) {
            vehicles[i].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.

```
class Book {
    String title;
    int publicationYear;

    Book(String title, int publicationYear) {
        this.title = title;
        this.publicationYear = publicationYear;
    }

    void displayInfo() {
        System.out.println("Book Title: " + title);
        System.out.println("Publication Year: " + publicationYear);
```

```
    }
}

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("Author Name: " + name);
        System.out.println("Author Bio: " + bio);
    }
}


public class LibraryManagement {
    public static void main(String[] args) {
        Author bookAuthor = new Author("The Great Gatsby", 1925, "F. Scott
Fitzgerald", "American novelist and short-story writer.");
        bookAuthor.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.

```
class Device {
    int deviceId;
    String status;

    Device(int deviceId, String status) {
        this.deviceId = deviceId;
        this.status = status;
    }

    void displayStatus() {
        System.out.println("Device ID: " + deviceId + ", Status: " +
status);
    }
}

class Thermostat extends Device {
    int temperatureSetting;

    Thermostat(int deviceId, String status, int temperatureSetting) {
        super(deviceId, status);
        this.temperatureSetting = temperatureSetting;
    }

    void displayStatus() {
        super.displayStatus();
        System.out.println("Temperature Setting: " + temperatureSetting +
"°C");
    }
}

public class SmartHomeSystem {
    public static void main(String[] args) {
        Thermostat thermostat = new Thermostat(101, "On", 24);
        thermostat.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.

```
class Order {
    int orderId;
    String orderDate;

    Order(int orderId, String orderDate) {
        this.orderId = orderId;
        this.orderDate = orderDate;
    }

    void getOrderStatus() {
        System.out.println("Order ID: " + orderId + " is placed on " +
orderDate);
```

```
    }
}
class ShippedOrder extends Order {
    String trackingNumber;

    ShippedOrder(int orderId, String orderDate, String trackingNumber) {
        super(orderId, orderDate);
        this.trackingNumber = trackingNumber;
    }

    void getOrderStatus() {
        System.out.println("Order ID: " + orderId + " is shipped with
Tracking Number: " + trackingNumber);
    }
}
class DeliveredOrder extends ShippedOrder {
    String deliveryDate;

    DeliveredOrder(int orderId, String orderDate, String trackingNumber,
String deliveryDate) {
        super(orderId, orderDate, trackingNumber);
        this.deliveryDate = deliveryDate;
    }

    void getOrderStatus() {
        System.out.println("Order ID: " + orderId + " was delivered on " +
deliveryDate);
    }
}

public class OnlineRetailOrder {
    public static void main(String[] args) {
        Order order = new Order(1001, "2024-03-01");
        ShippedOrder shippedOrder = new ShippedOrder(1002, "2024-03-02",
"TRK12345");
        DeliveredOrder deliveredOrder = new DeliveredOrder(1003,
"2024-03-03", "TRK67890", "2024-03-05");

        order.getOrderStatus();
        shippedOrder.getOrderStatus();
        deliveredOrder.getOrderStatus();
```

```
    }
}
```

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.

```
class Course {
    String courseName;
    int duration;

    Course(String courseName, int duration) {
        this.courseName = courseName;
        this.duration = duration;
    }

    void displayInfo() {
        System.out.println("Course Name: " + courseName);
        System.out.println("Duration: " + duration + " weeks");
    }
}
class OnlineCourse extends Course {
    String platform;
    boolean isRecorded;
```

```
    OnlineCourse(String courseName, int duration, String platform, boolean
isRecorded) {
        super(courseName, duration);
        this.platform = platform;
        this.isRecorded = isRecorded;
    }

    void displayInfo() {
        super.displayInfo();
        System.out.println("Platform: " + platform);
        System.out.println("Recorded: " + (isRecorded ? "Yes" : "No"));
    }
}
class PaidOnlineCourse extends OnlineCourse {
    double fee;
    double discount;

    PaidOnlineCourse(String courseName, int duration, String platform,
boolean isRecorded, double fee, double discount) {
        super(courseName, duration, platform, isRecorded);
        this.fee = fee;
        this.discount = discount;
    }

    void displayInfo() {
        super.displayInfo();
        System.out.println("Course Fee: $" + fee);
        System.out.println("Discount: " + discount + "%");
    }
}

public class EducationalCourseHierarchy {
    public static void main(String[] args) {
        PaidOnlineCourse poc = new PaidOnlineCourse("Java Programming", 8,
"Udemy", true, 100, 20);
        poc.displayInfo();
    }
}
```

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.

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.

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

    Person(String name, int age) {
```

```
        this.name = name;
        this.age = age;
    }
}
class Teacher extends Person {
    String subject;

    Teacher(String name, int age, String subject) {
        super(name, age);
        this.subject = subject;
    }
    void displayRole() {
        System.out.println(name + " is a teacher of " + subject);
    }
}

class Student extends Person {
    int grade;

    Student(String name, int age, int grade) {
        super(name, age);
        this.grade = grade;
    }
    void displayRole() {
        System.out.println(name + " is a student in grade " + grade);
    }
}

class Staff extends Person {
    String department;

    Staff(String name, int age, String department) {
        super(name, age);
        this.department = department;
    }
    void displayRole() {
        System.out.println(name + " is a staff member in " + department);
    }
}

public class SchoolSystem {
```

```
    public static void main(String[] args) {
        Teacher teacher = new Teacher("Alice", 35, "Mathematics");
        Student student = new Student("Bob", 15, 10);
        Staff staff = new Staff("Charlie", 40, "Administration");

        teacher.displayRole();
        student.displayRole();
        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.

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

```java
class Person1 {
    String name;
    int id;

    Person1(String name, int id) {
        this.name = name;
        this.id = id;
    }
}

class Chef extends Person1 implements Worker {
    Chef(String name, int id) {
        super(name, id);
    }
    public void performDuties() {
        System.out.println(name + " is cooking food.");
    }
}

class Waiter extends Person1 implements Worker {
    Waiter(String name, int id) {
        super(name, id);
    }
    public void performDuties() {
        System.out.println(name + " is serving customers.");
    }
}

public class RestaurantSystem {
    public static void main(String[] args) {
        Chef chef = new Chef("John", 101);
        Waiter waiter = new Waiter("Mark", 202);

        chef.performDuties();
        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.

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

class Vehicle {
    int maxSpeed;
    String model;

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

class ElectricVehicle extends Vehicle {
    ElectricVehicle(int maxSpeed, String model) {
        super(maxSpeed, model);
    }
    void charge() {
        System.out.println(model + " is charging.");
    }
}
```

```java
class PetrolVehicle extends Vehicle implements Refuelable {
    PetrolVehicle(int maxSpeed, String model) {
        super(maxSpeed, model);
    }
    public void refuel() {
        System.out.println(model + " is refueling.");
    }
}

public class VehicleManagement {
    public static void main(String[] args) {
        ElectricVehicle ev = new ElectricVehicle(150, "Tesla Model 3");
        PetrolVehicle pv = new PetrolVehicle(200, "Ford Mustang");

        ev.charge();
        pv.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.