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.
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.
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.

CodInClub
Powered By BridgeLabz

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.

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.

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.

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.

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.

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.

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.

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.