Module 4 Inheritance and Polymorphism

Premanand S

Assistant Professor
School of Electonics Engineering
Vellore Institute of Technology
Chennai Campus

premanand.s@vit.ac.in

March 3, 2025

Contents - Module 3

- Inheritance
- Polymorphism

Inheritance

Inheritance

- Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows a class (child/subclass) to acquire the properties and behaviors of another class (parent/superclass).
- It promotes code reusability and hierarchical classification of objects.

Inheritance Syntax - Example

Example (Understanding)

```
class Parent {
    void display() {
        System.out.println("This is a Parent class method.");
    }}
class Child extends Parent {
    void show() {
        System.out.println("This is a Child class method.");
    }}
public class Main {
    public static void main(String[] args) {
        Child obj = new Child();
        obj.display(); // Inherited from Parent
        obj.show();    // Child class method
    }}
```

Real-time examples of Inheritance

- Parent-Child Relationship
- Vehicles (Hierarchy Example)
  - Base Class (Vehicle): Contains common properties like speed, fuelType
  - Derived Classes (Car, Bike, Truck): Inherits properties of Vehicle but also has unique features like numOfWheels or cargoCapacity

Inheritance- Example

Example (Understanding)

```
class Vehicle {
    String fuelType = "Petrol";

    void start() {
        System.out.println("Vehicle is starting...");
    }}
class Car extends Vehicle {
    int numOfWheels = 4;
    void showDetails() {
        System.out.println("Car has " + numOfWheels +
        " wheels and runs on " + fuelType);
    }}
```

Inheritance - Example

Example (Understanding)

```
public class Test {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.start();  // Inherited from Vehicle
        myCar.showDetails();
    }}
```

Inheritance - Types

Why Do We Need Inheritance in Java?

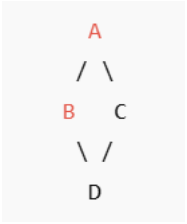
- Code Reusability
- Extensibility
- Readability & Maintainability
- Reduces Code Redundancy
- Enhances Polymorphism

Types of Inheritance

- Inheritance in Java allows a class to derive properties and behaviors from another class
    - Single Inheritance
    - Multilevel Inheritance
    - Hierarchical Inheritance
    - Multiple Inheritance (via Interfaces)
    - Hybrid Inheritance (Combination)
- But Java does not support multiple inheritance using classes due to the Diamond Problem. However, multiple inheritance is possible with interfaces

Diamond Problem?

Diamond problem?



- Class A has a method show().
- Class B and Class C inherit from Class A and override show().
- Class D inherits from both B and C.
- If D calls show(), which version should be used? B's or C's?

Single Inheritance

Single Inheritance

- Simplest form of inheritance where a child class (subclass) inherits from one parent class (superclass).
- Key Points,
    - One parent, one child
    - Child class reuses the properties and methods of the parent
    - Supports code reusability and maintainability
    - Uses the extends keyword

Single Inheritance Syntax

Example (Understanding)

```
class Parent {
    // Parent class properties and methods
}

class Child extends Parent {
    // Child class can access Parent class
    properties and methods
}
```

Single Inheritance - Example

Example (Understanding)

```
// Superclass (Parent)
class Animal {
    void eat() {
        System.out.println("I can eat");
    }
}
```

Single Inheritance - Example

Example (Understanding)

```
// Subclass (Child)
class Dog extends Animal {
    void bark() {
        System.out.println("I can bark");
    }
}
```

Single Inheritance - Example

Example (Understanding)

```
// Main class
public class SingleInheritanceDemo {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();  // Inherited method from Animal class
        myDog.bark(); // Method from Dog class
    }
}
```

Important Concepts in Single Inheritance

- super Keyword – Used to call parent class methods or constructors
- Method Overriding – Child class can override parent class methods
- Access Modifiers – Controls visibility of inherited methods (public, protected, private)

super Keyword - Single Inheritance

- The super keyword is used in a child class to refer to its immediate parent class. It is mainly used for:
  - Calling parent class methods
  - Calling parent class constructors
  - Accessing parent class variables

super - Parent class

Using super to Call Parent Class Method

Example (Understanding)

```
class Animal {
    void display() {
        System.out.println("I am an Animal");
    }
}
```

Using super to Call Parent Class Method

Example (Understanding)

```
class Dog extends Animal {
    void display() {
        super.display(); // Calls Parent class method
        System.out.println("I am a Dog");
    }
}
```

Using super to Call Parent Class Method

Example (Understanding)

```
public class SuperMethodDemo {
    public static void main(String[] args) {
        Dog obj = new Dog();
        obj.display();
    }
}
```

super - parent constructor

Using super to Call Parent Constructor

Example (Understanding)

```
class Animal {
    Animal() {
        System.out.println("Animal Constructor");
    }
}
```

Using super to Call Parent Constructor

Example (Understanding)

```
class Dog extends Animal {
    Dog() {
        super();  // Calls Parent class constructor
        System.out.println("Dog Constructor");
    }
}
```

Using super to Call Parent Constructor

Example (Understanding)

```
public class SuperConstructorDemo {
    public static void main(String[] args) {
        Dog obj = new Dog();
    }
}
```

method overriding

Method Overriding

- Method Overriding allows a child class to provide a specific implementation of a method already defined in its parent class.
  - Same method name in parent and child
  - Same parameters and return type
  - Overridden method must be public or protected (not private)

Overriding Parent Method

Example (Understanding)

```
class Animal {
    void sound() {
        System.out.println("Animals make sounds");
    }
}
```

Overriding Parent Method

Example (Understanding)

```
class Dog extends Animal {
    @Override
    void sound() {  // Overriding Parent method
        System.out.println("Dogs bark");
    }
}
```

Overriding Parent Method

Example (Understanding)

```
public class MethodOverridingDemo {
    public static void main(String[] args) {
        Dog obj = new Dog();
        obj.sound();  // Calls overridden method in Dog class
    }
}
```

super keyword vs Method Overriding

- When to Use super?
  - Calling Parent Methods: If you need to execute both parent child versions of a method
  - Calling Parent Constructor: If the child constructor must initialize parent attributes
  - Accessing Hidden Parent Variables: If a child class defines a variable with the same name as the parent
- When to Use Method Overriding?
  - To Modify Behavior: When the parent method does not fit the child class
  - For Dynamic Method Dispatch (Polymorphism): When a parent reference calls a method of a child
  - For Abstract Methods: If the parent class has an abstract method, the child must override it

Multilevel Inheritance

Multilevel Inheritance

- A type of inheritance where a child class inherits from a parent class, and another class further extends this child class.
- Characteristics
  - A child class gets the properties of all its ancestors
  - The superclass of one class acts as a subclass of another
  - Supports code reusability
  - Can lead to complexity if overused

Multilevel Inheritance - example

Example (Understanding)

```
// Parent Class (Base Class)
class Animal {
    void eat() {
        System.out.println("Animals eat food");
    }
}
```

Multilevel Inheritance - example

Example (Understanding)

```
// Intermediate Class (Child of Animal, Parent of Dog)
class Mammal extends Animal {
    void walk() {
        System.out.println("Mammals walk on land");
    }
}
```

Multilevel Inheritance - example

Example (Understanding)

```
// Child Class (Derived Class)
class Dog extends Mammal {
    void bark() {
        System.out.println("Dogs bark");
    }
}
```

Multilevel Inheritance - example

Example (Understanding)

```
// Main Class
public class MultilevelInheritanceDemo {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();  // Inherited from Animal
        d.walk(); // Inherited from Mammal
        d.bark(); // Defined in Dog
    }
}
```

Multilevel Inheritance - How it works?

- Animal class $\rightarrow$ Base Class
- Mammal class inherits Animal
- Dog class inherits Mammal
- Dog can access methods from both Mammal and Animal

Using super in Multilevel Inheritance

Example (Understanding)

```
class Animal {
    Animal() {
        System.out.println("Animal Constructor");
    }
}

class Mammal extends Animal {
    Mammal() {
        super();  // Calls Animal constructor
        System.out.println("Mammal Constructor");
    }
}
```

Using super in Multilevel Inheritance

Example (Understanding)

```
class Dog extends Mammal {
    Dog() {
        super();  // Calls Mammal constructor
        System.out.println("Dog Constructor");
    }
}

public class ConstructorChainingDemo {
    public static void main(String[] args) {
        Dog d = new Dog();
    }
}
```

Using super in Multilevel Inheritance

- super(); ensures all ancestor constructors are called in the correct order.
- The execution starts from the top (Animal) and flows down to Dog.

Method Overriding in Multilevel Inheritance

- Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. In multilevel inheritance, overriding can be done at multiple levels.
- Rules,
  - Same method signature in both parent and child class
  - The child class method must not reduce the access level (e.g., public in parent $\rightarrow$ must remain public in child)
  - Can use super.methodName() to call the overridden method of the parent class
  - Supports runtime polymorphism

Example 1: Overriding in Multilevel Inheritance

Example (Understanding)

```
// Base Class
class Vehicle {
    void move() {
        System.out.println("Vehicles can move");
    }
}

// Intermediate Class
class Car extends Vehicle {
    @Override
    void move() {
        System.out.println("Cars move on roads");
    }
}
```

Example 1: Overriding in Multilevel Inheritance

Example (Understanding)

```
// Derived Class
class ElectricCar extends Car {
    @Override
    void move() {
        System.out.println("Electric Cars move silently");
    }
}

// Main Class
public class OverrideMultilevelDemo {
    public static void main(String[] args) {
        ElectricCar tesla = new ElectricCar();
        tesla.move();  // Calls overridden method from Electri
    }
}
```

Example 2: Using super to Call Parent Methods

Example (Understanding)

```
class Vehicle {
    void move() {
        System.out.println("Vehicles can move");
    }
}

class Car extends Vehicle {
    @Override
    void move() {
        super.move();  // Calls Vehicle's move() method
        System.out.println("Cars move on roads");
    }
}
```

Example 2: Using super to Call Parent Methods

Example (Understanding)

```
class ElectricCar extends Car {
    @Override
    void move() {
        super.move();  // Calls Car's move() method
        System.out.println("Electric Cars move silently");
    }
}

public class SuperOverrideDemo {
    public static void main(String[] args) {
        ElectricCar tesla = new ElectricCar();
        tesla.move();  // Calls move() method from all levels
    }
}
```

final keyword

final Keyword and Preventing Overriding in Java

- The final keyword in Java is used to restrict modifications to variables, methods, and classes. When applied to methods, it prevents method overriding in subclasses.
- If a method is declared as final, it cannot be overridden by any subclass.
- This ensures that the method's implementation remains unchanged in all child classes.

Example: Preventing Method Overriding

Example (Understanding)

```
class Person {
    final void work() {
        System.out.println("People do different types of work'
    }
}

class Employee extends Person {
    //  Compilation Error! Cannot override a final method
    /*
    void work() {
        System.out.println("Employees work in offices");
    }
    */
}
```

Example: Preventing Method Overriding

Example (Understanding)

```
public class FinalMethodDemo {
    public static void main(String[] args) {
        Employee e = new Employee();
        e.work();  // Calls Person’s work() method
    }
}
```

Example: Preventing Inheritance

Example (Understanding)

```
final class Animal {
    void sound() {
        System.out.println("Animals make sounds");
    }
}

//  Compilation Error! Cannot extend a final class
/*
class Dog extends Animal {
    void sound() {
        System.out.println("Dogs bark");
    }
}
*/
```

Example: Preventing Inheritance

Example (Understanding)

```
public class FinalClassDemo {
    public static void main(String[] args) {
        Animal a = new Animal();
        a.sound();  // Works fine
    }
}
```

Example: Final Variable

Example (Understanding)

```
class Constants {
    final double PI = 3.14159;

    void changeValue() {
        //  Compilation Error! Cannot reassign a final variabl
        // PI = 3.14;
    }
}
```

Hierarchical Inheritance

Hierarchical Inheritance

- In Hierarchical Inheritance, multiple child classes inherit from a single parent class.
- The parent class acts as a common base for all child classes.

Example1 of Hierarchical Inheritance

Example (Understanding)

```
// Parent Class
class Animal {
    void sound() {
        System.out.println("Animals make sounds");
    }
}

// Child Class 1
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}
```

Example1 of Hierarchical Inheritance

Example (Understanding)

```
// Child Class 2
class Cat extends Animal {
    void meow() {
        System.out.println("Cat meows");
    }
}

// Main Class
public class HierarchicalInheritanceDemo {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();  // Inherited from Animal
        d.bark();

        Cat c = new Cat();
```

Example2 of Hierarchical Inheritance

Example (Understanding)

```
// Parent Class
class Vehicle {
    void start() {
        System.out.println("Vehicle is starting");
    }
}

// Child Class 1
class Car extends Vehicle {
    void fuelType() {
        System.out.println("Car runs on petrol or diesel");
    }
}
```

Example2 of Hierarchical Inheritance

Example (Understanding)

```
// Child Class 2
class Bike extends Vehicle {
    void twoWheeler() {
        System.out.println("Bike has two wheels");
    }
}
public class HierarchicalExample {
    public static void main(String[] args) {
        Car c = new Car();
        c.start();    // Inherited from Vehicle
        c.fuelType();
        Bike b = new Bike();
        b.start();    // Inherited from Vehicle
        b.twoWheeler();
    } }
```

Multiple Inheritance

Multiple Inheritance

- Multiple Inheritance allows a class to inherit from more than one parent class.
- Java does NOT support multiple inheritance with classes to avoid ambiguity issues (Diamond Problem).
- However, multiple inheritance is possible using interfaces.

Why Doesn't Java Support Multiple Inheritance with Classes?

Example (Understanding)

```
class A {
public:
    void show() { cout << "A's show" << endl; }
};
class B {
public:
    void show() { cout << "B's show" << endl; }
};
```

Why Doesn't Java Support Multiple Inheritance with Classes?

Example (Understanding)

```
//  Problem: Class C inherits from both A and B
class C : public A, public B {};
int main() {
    C obj;
    obj.show();  //  Ambiguity: Should it call
    A's show() or B's show()?
}
```

Problem: Ambiguity

- C inherits show() from both A and B.
- When C calls show(), which version should be used? (A's or B's?)
- Java avoids this issue by disallowing multiple inheritance with classes.

How Does Java Allow Multiple Inheritance?

- Classes cannot inherit from multiple classes (to avoid ambiguity).
- But a class can implement multiple interfaces, as interfaces do not store implementation, only method signatures.
- If two interfaces have the same method, the implementing class must override it to avoid ambiguity.

Multiple Inheritance Using Interfaces

Example (Understanding)

```
// First Interface
interface Animal {
    void makeSound();
}

// Second Interface
interface Pet {
    void play();
}
```

Multiple Inheritance Using Interfaces

Example (Understanding)

```
// A class implementing both interfaces
class Dog implements Animal, Pet {
    public void makeSound() {
        System.out.println("Dog barks");
    }

    public void play() {
        System.out.println("Dog plays with a ball");
    }
}
```

Multiple Inheritance Using Interfaces

Example (Understanding)

```
public class MultipleInheritanceExample {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();
        myDog.play();
    }
}
```

Handling Method Conflicts in Multiple Interfaces

- If two interfaces have the same default method, Java forces the implementing class to override the method to resolve ambiguity.

Example (Understanding)

```
interface A {
    default void show() {
        System.out.println("A’s show");
    }
}
```

Handling Method Conflicts in Multiple Interfaces

Example (Understanding)

```
interface B {
    default void show() {
        System.out.println("B’s show");
    }
}

// Class implementing both interfaces
class C implements A, B {
    // Overriding to resolve ambiguity
    public void show() {
        System.out.println("C’s own show method");
    }
}
```

Handling Method Conflicts in Multiple Interfaces

Example (Understanding)

```
public class MultipleInheritanceConflict {
    public static void main(String[] args) {
        C obj = new C();
        obj.show();  // Calls C's overridden method
    }
}
```

Hybrid Inheritance

Hybrid Inheritance

- Hybrid Inheritance is a combination of two or more types of inheritance (Single, Multilevel, Hierarchical, or Multiple).
- Java does NOT support hybrid inheritance with classes due to the Diamond Problem.
- However, Hybrid Inheritance can be implemented using interfaces.

Why Doesn't Java Support Hybrid Inheritance with Classes?

Example (Understanding)

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

class B extends A {} // B inherits A
class C extends A {} // C inherits A

//  Compilation Error: Java does not allow multiple inheritanc
class D extends B, C { // D inherits from both B and C
    // Ambiguity: Should it call show() from B or C?
}
```

Why Doesn't Java Support Hybrid Inheritance with Classes?

Example (Understanding)

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

class B extends A {} // B inherits A
class C extends A {} // C inherits A

//  Compilation Error: Java does not allow multiple inheritanc
class D extends B, C { // D inherits from both B and C
    // Ambiguity: Should it call show() from B or C?
}
```

Why Doesn't Java Support Hybrid Inheritance with Classes?

Example (Understanding)

```
public class HybridInheritanceError {
    public static void main(String[] args) {
        D obj = new D();
        obj.show();  //  Error: Ambiguity in Java
    }
}
```

Why Doesn't Java Support Hybrid Inheritance with Classes?

Example (Understanding)

```
public class HybridInheritanceError {
    public static void main(String[] args) {
        D obj = new D();
        obj.show();  //  Error: Ambiguity in Java
    }
}
```

Correct Approach: Using Interfaces

Example (Understanding)

```
// Interface A with a default method
interface A {
    default void show() {
        System.out.println("A’s show method");
    }
}

// Interface B extends A
interface B extends A {}

// Interface C extends A
interface C extends A {}
```

Correct Approach: Using Interfaces

Example (Understanding)

```
// Class D implements both B and C
class D implements B, C {
    // Resolving ambiguity by overriding the show() method
    public void show() {
        System.out.println("D’s own show method");
    }
}

public class HybridInheritanceSolution {
    public static void main(String[] args) {
        D obj = new D();
        obj.show();  // Calls D’s overridden method
    }
}
```

Hybrid Inheritance - takeaway

- Java does NOT support multiple inheritance with classes (to avoid the Diamond Problem).
- Java supports multiple inheritance using interfaces, as they do not store method implementations.
- If two interfaces have the same method, the implementing class must override it to avoid ambiguity.