Inheritance in Java (OOP's Method)

1. What is Inheritance? (Definition)

Inheritance is one of the core concepts of Object-Oriented Programming (OOP). Think of it like this: just as children inherit traits and characteristics from their parents, in Java, a child class can inherit properties and methods from a parent class. This means you can create a new class that automatically gets all the useful features of an existing class without writing the code again.

In simple terms, inheritance allows one class to reuse and extend the functionality of another class. The class that provides the features is called the parent class (or superclass), and the class that gets those features is called the child class (or subclass).

Technical Definition: Inheritance is a mechanism in Java that enables one class to inherit the properties and methods of another class, establishing an "is-a" relationship between them.

2. Why is Inheritance Important? (Purpose & Benefits)

Inheritance is one of the most valuable features of object-oriented programming. Here's why you should care about it:

Reason 1: Reusability (Write Once, Use Many Times)

Imagine you're building a system for an online store. You have different types of products: Electronics, Books, Clothing, etc. All of them have common features like:

- Product name
- Price
- Stock quantity
- Method to display details
- Method to calculate discount

Without inheritance, you'd write all this code multiple times in every product class. With inheritance, you write it once in a parent class and all product types automatically get it. This saves you hours of work!

Reason 2: Easier Maintenance (Change Once, Update Everywhere)

Suppose you discover a bug in the discount calculation method. Without inheritance, you'd have to fix it in every single class. With inheritance, you fix it once in the parent class, and all child classes automatically get the fix. This is a huge time-saver and reduces mistakes.

Reason 3: Logical Organization (Real-World Structure)

Inheritance lets you organize your code the way the real world works. For example:

- Vehicle is the parent
  - Car is a type of Vehicle
  - Motorcycle is a type of Vehicle
  - Truck is a type of Vehicle

This makes your code easier to understand and maintain because it mirrors how we think about things in reality.

Reason 4: Polymorphism (Flexibility & Power)

Inheritance enables polymorphism, which means different classes can respond to the same method call in different ways. For instance, both a Dog and a Cat can have a `sound()'` method, but Dog will bark and Cat will meow. This flexibility is powerful for building scalable systems.

3. When Should You Use Inheritance? (Real-World Scenarios)

Inheritance should be used when there's a clear, logical relationship between classes. Here's when to use it and when to avoid it:

Use Inheritance When:

- There's an "is-a" relationship : A Dog is-an Animal, a Student is-a Person
- Multiple classes share common properties and behaviors
- You want to extend an existing class without modifying it
- You need polymorphism (different implementations of the same method)

Don't Use Inheritance When:

- There's a "has-a" relationship : A Car has-an Engine (use composition instead)

- The parent class is marked as Final (you can't extend Final classes)
- You just want to use features from another class without a logical connection
- The relationship is temporary or changing

Real Example :

- Use inheritance: Employee > Manager, Developer, Designer
- Don't use: Car extends Engine (wrong! A Car has-an Engine, so use composition)

---

4. How Does Inheritance Work? (Technical Implementation)

Step-by-Step Working

1. The child class gets access to all public and protected members of the parent
2. The child class can use all parent methods as if they were its own
3. The child class can add its own new methods
4. The child class can override (rewrite) parent methods with its own versions

Basic Syntax

```
// Parent class
class ParentClass {
    int parentVariable;

    void parentMethod() {
        System.out.println("This is parent method");
    }
}

// Child class inherits from parent using 'extends' keyword
class ChildClass extends ParentClass {
    int childVariable;

    void childMethod() {
        System.out.println("This is child method");
    }
}
```

The keyword `extends` is what makes the magic happen. It tells Java: "ChildClass should inherit everything from ParentClass."

Here's Example

Let's build a Parent and Child Program to understand inheritance better:

```
// A Simple Java Program in Inheritance Concept between Mom(Parent) and Daughter(Child).

class Mom {

    String hairColour = "Black";
    String eyeColour = "Brown";
    void cook() {
        System.out.println("Mom is Cooking Dishes");
    }
    void work() {
        System.out.println("Mom is Working.");
    }
}
// Child class inherits from parent using 'extends' keyword
class Daughter extends Mom {
    String hobby = "Dancing";
    int age = 18;
    void study() {
        System.out.println("Daughter is Studying .");
    }
}
public class Inheritance_4 {
    public static void main(String[] args) {

        // Mom Own Properties

        Mom Kalpana = new Mom();
        Kalpana.work();
        Kalpana.cook();
        System.out.println("Mom's Hair Colour : " + Kalpana.hairColour);
        System.out.println("Mom's Eye  Colour : " + Kalpana.eyeColour);

        // Daughter Own Properties

        System.out.println("--------Daughter Own Properties--------");
        Daughter Reena = new Daughter();
        Reena.study();
        System.out.println("Daughter's Hobby : " + Reena.hobby);
        System.out.println("Daughter's Age : " + Reena.age);
```

```
        // Daughter Inherited Mom's Properties

        System.out.println("--------Daughter Inherited Mom's Properties--------");
        System.out.println("Daughter's Hair Colour : " + Reena.hairColour);
        System.out.println("Daughter's Eye Colour : " + Reena.eyeColour);
        System.out.println("Reena's Mom Actions are:");
        Reena.cook();
        Reena.work();
    }
}
```

```
Output :-
Mom is Working.
Mom is Cooking Dishes
Mom's Hair Colour : Black
Mom's Eye  Colour : Brown
--------Daughter Own Properties--------
Daughter is Studying .
Daughter's Hobby : Dancing
Daughter's Age : 18
--------Daughter Inherited Mom's Properties--------
Daughter's Hair Colour : Black
Daughter's Eye Colour : Brown
Reena's Mom Actions are:
Mom is Cooking Dishes
Mom is Working.
```

Understanding Access Modifiers in Inheritance

When you inherit from a parent class, not all members are accessible. It depends on their access modifier:

Modifier	Child Class Access	Same Package	Different Package
public	✅ Yes	✅ Yes	✅ Yes
protected	✅ Yes	✅ Yes	❌ No
default	✅ Yes	✅ Yes	❌ No
private	❌ No	❌ No	❌ No

Important: Private members are NOT inherited, but you can access them through public getter methods.

5. Types of Inheritance in Java

Java supports five different types of inheritance. Let's understand each one with real examples:

Type 1: Single Inheritance (One Parent, One Child)

**Single inheritance is the simplest form where one child class extends exactly one parent class.

```
    Vehicle (Parent)
       ↑
       |
    Car (Child)
```

Real-World Scenario: Every car is a vehicle, so Car should inherit from Vehicle.

Type 2: Multilevel Inheritance (A Chain of Classes)

In multilevel inheritance, a class inherits from another class which itself inherits from a third class. It forms a chain.

```
    Animal (Grandparent)
        ↑
        |
    Mammal (Parent)
        ↑
        |
      Dog (Child)
```

Real-World Scenario: Dogs are mammals, and mammals are animals.

Type 3: Hierarchical Inheritance (One Parent, Multiple Children)

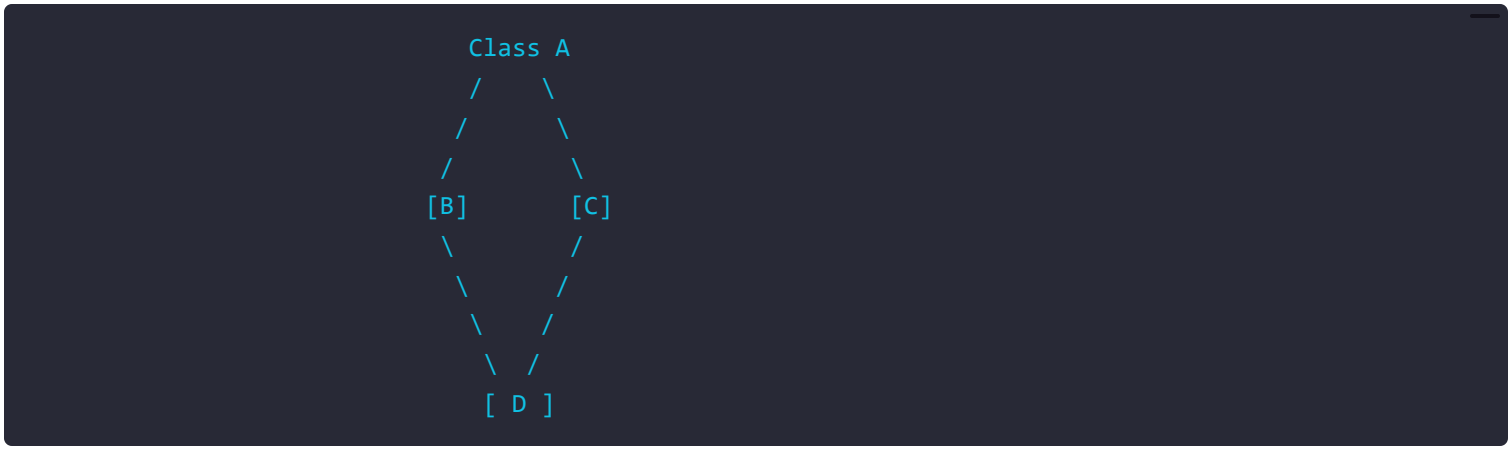
Multiple child classes inherit from the same parent class.

Real-World Scenario: Different types of employees (Manager, Developer, Designer) all share common employee features.

Type 4: Multiple Inheritance (NOT Directly Supported)

Multiple inheritance means a class inheriting from multiple parent classes. Java does NOT directly support this because it causes the "Diamond Problem."

The Diamond Problem:



If D inherits from both B and C, and both B and C have a method doSomething(), which version should D use? This confusion is why Java doesn't allow it.

Type 5: Hybrid Inheritance (Mix of Multiple Types)

Hybrid inheritance combines multiple types of inheritance within a single program. This is common in real applications.

Example Structure:

```
        Vehicle (Base class)
            ↑
        ____|____
       |         |
     Car        Bike (Hierarchical)
      ↑
      |
ElectricCar     (Single)
```

This combines both hierarchical and single inheritance.

---

6. Method Overriding (Customizing Parent Behavior)

Method overriding occurs when a child class provides its own implementation of a method that already exists in the parent class. This is how you customize behavior for different types.

Why Override Methods?

Sometimes the parent's method is too general. You want child classes to have their own specialized versions.

Example: A parent class Shape has a method getArea(). But a Circle calculates area differently than a Rectangle. So each child class overrides getArea() with its own calculation.

Rules for Method Overriding

To properly override a method, follow these rules:

1. Method name must be identical to the parent's method.
2. Parameters must be the same (same types and number).
3. Return type must be the same or a compatible type.
4. Access modifier can be same or more accessible (e.g., can change from protected to public, but not vice versa).
5. Should use `@Override` annotation to make intent clear.

**Example :-

```java
class Animal {
    public void sound() {
        System.out.println("Animal makes some sound");
    }
}

class Dog extends Animal {
```

```
    @Override // Indicates this method overrides parent method
    public void sound() {
        System.out.println("Dog barks: Woof Woof!");
    }
}

class Cat extends Animal {
    @Override
    public void sound() {
        System.out.println("Cat meows: Meow Meow!");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        animal.sound(); // Output: Animal makes some sound

        Dog dog = new Dog();
        dog.sound();  // Output: Dog barks: Woof Woof!

        Cat cat = new Cat();
        cat.sound();  // Output: Cat meows: Meow Meow!
    }
}
```

7. The super Keyword (Accessing Parent Class)

The super keyword allows you to explicitly refer to members of the parent class from within the child class. It's useful when you want to use or extend parent functionality rather than completely replace it.

Use Case 1: Calling Parent Constructor

When creating a child object, you often need to initialize parent class fields. Use super() to call the parent constructor:

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

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

```
        this.age = age;
    }
}

class Student extends Person {
    String rollNo;

    Student(String name, int age, String rollNo) {
        super(name, age);  // Calls parent constructor
        this.rollNo = rollNo;
    }
}

public class Main {
    public static void main(String[] args) {
        Student student = new Student("Badal", 18, "CS001");
        System.out.println("Name: " + student.name);
        System.out.println("Age: " + student.age);
        System.out.println("Roll No: " + student.rollNo);
    }
}
```

```
Output:-
 Name: Badal
 Age: 18
 Roll No: CS001
```

Use Case 2: Calling Parent Method

When you override a method but still want to use the parent's version along with your customization, use `super.methodName()`:

```
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        super.sound(); // Calls parent method
        System.out.println("Dog specifically barks");
    }
}

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

```
        Dog dog = new Dog();
        dog.sound();
    }
}
```

```
Output:
Animal makes sound
Dog specifically barks
```

Use Case 3: Accessing Parent Fields

Sometimes you need to access a parent class field that's marked as protected:

```
class Vehicle {
    protected String brand;
}

class Car extends Vehicle {
    String model;

    void display() {
        System.out.println("Brand: " + super.brand);  // Access parent field
        System.out.println("Model: " + model);
    }
}
```

8. Advantages of Using Inheritance

Advantage	Explanation
Code Reusability	Write code once in parent, use in multiple child classes
Easy Maintenance	Update parent once, all children benefit automatically
Logical Organization	Code organized in a hierarchical structure that mirrors reality
Polymorphism	Different child classes can have different implementations of same method
Less Code Duplication	Avoid writing same code multiple times
Flexibility	Easy to add new child classes without modifying existing code

DRY Principle	"Don't Repeat Yourself" - inheritance helps follow this principle

9. Disadvantages of Inheritance

Disadvantage	Why It's a Problem	Solution
Tight Coupling	Parent and child classes become dependent on each other; changing parent might break children	Design carefully; keep relationships loose
Complexity	Deep hierarchies become hard to understand and maintain	Keep inheritance chains shallow (max 3 levels)
Inflexible	Inheritance is decided at compile-time and can't change at runtime	Use interfaces and composition for more flexibility
Fragile Base Class Problem	Changes to parent class can unexpectedly break child classes	Be careful when modifying parent class methods
Limited by Access Modifiers	Private members in parent aren't inherited	Design public/protected interfaces carefully

Java Inheritance Method Access Rules

Case	Reference Type	Object Type	Result	Reason
Parent p = new Parent ()	Parent	Parent	✅ Valid	Parent apni methods access kar sakta hai
Child c = new Child ()	Child	Child	✅ Valid	Child apni + inherited methods access kar sakti hai
Parent p = new Child ()	Parent	Child	⚠️ Partial	Upcasting - Sirf Parent ki methods accessible hain
Child c = new Parent ()	Child	Parent	❌ Error	Type incompatible - Parent IS-NOT-A Child

Example :-

Practical Example - Complete Code

```java
class Mom {
    String hairColor = "Black";

    void cook() {
        System.out.println("Mom cooking kar rahi hai");
    }

    void work() {
        System.out.println("Mom office mein kaam kar rahi hai");
    }
}

class Daughter extends Mom {
    String hobby = "Dancing";

    void study() {
        System.out.println("Daughter padh rahi hai");
    }
}

public class Main {
    public static void main(String[] args) {
        // CASE 1
        System.out.println("--- CASE 1 ---");
        Mom m1 = new Mom();
        m1.cook();      // ✅
        m1.work();      // ✅

        // CASE 2
        System.out.println("\n--- CASE 2 ---");
        Daughter d2 = new Daughter();
        d2.study();     // ✅
        d2.cook();      // ✅ (inherited)
        d2.work();      // ✅ (inherited)

        // CASE 3 - Upcasting
        System.out.println("\n--- CASE 3 ---");
        Mom m3 = new Daughter();  // Upcasting
        m3.cook();      // ✅
        m3.work();      // ✅
        // m3.study();  // ❌ Compile Error

        // CASE 4 - Not allowed
        // Daughter d4 = new Mom();  // ❌ Compile Error
    }
}
```