Inheritance

▼ Introduction to Inheritance in Java

What is Inheritance?

Inheritance is one of the **core principles of Object-Oriented Programming (OOP)** that allows a new class (called a **subclass** or **child class**) to **inherit** properties and methods from an existing class (called a **superclass** or **parent class**). This enables the child class to **reuse** code from the parent class, thus promoting code reusability, scalability, and organization.

Why is Inheritance Used?

1. **Maintainability**: Changes made to the superclass automatically propagate to subclasses, reducing redundancy and effort when making updates.
2. **Flexibility**: You can add new functionality to existing classes without modifying the original class.
3. **Modularity**: The system can be modularized into smaller, logically related units (superclasses and subclasses), improving overall design and structure.
4. **Improved Code Readability**: Hierarchical class relationships help make code easier to understand and maintain, especially in large systems.

When to Use Inheritance?

1. **Is-A Relationship**: Use inheritance when there is an **"is-a"** relationship between the parent and child classes. For example, a **Dog** is an **Animal**, so **Dog** can inherit from **Animal**.
   - Example: `Dog extends Animal`
2. **When you want to extend the behavior of an existing class**: If you have a class and you want to create a new class that shares some common behaviors but also has additional or specialized behaviors, inheritance is ideal.

- Example: A `Vehicle` class could have subclasses like `Car`, `Truck`, etc., each extending the base properties of `Vehicle` but adding their own specific features.

3. **Code Refactoring**: When you have duplicate code, inheritance helps in creating a **generalized superclass** that all subclasses can inherit from, ensuring that the common code is written once.

How is Inheritance Used in Java?

In Java, inheritance is implemented using the `extends` keyword. The subclass inherits all non-private fields and methods of the superclass.

1. **Basic Syntax:**

```
class Superclass {
    // Superclass fields and methods
}

class Subclass extends Superclass {
    // Subclass-specific fields and methods
}
```

2. **Accessing Parent Class Members**: The subclass automatically inherits all the **non-private** members (fields and methods) of the superclass, but it cannot access **private members** directly. It can access them through public/protected getters and setters or constructors.

3. **Method Overriding**: A subclass can provide its own implementation of a method that is already defined in the superclass. This is known as **method overriding** and helps to modify or extend the behavior of inherited methods.

4. **Constructors**: A subclass can call the superclass constructor using the `super()` keyword. If not explicitly called, the default constructor of the superclass is automatically invoked.

5. **Polymorphism**: A reference variable of the superclass type can point to an object of the subclass. This allows the program to invoke overridden

methods and treat different objects in a uniform way.

Example: Simple Inheritance

```
class Animal {
    String name;

    void eat() {
        System.out.println("This animal eats food.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Inherited method
        dog.bark(); // Subclass method
    }
}
```

Output:

```
This animal eats food.
The dog barks.
```

In this example:

- `Dog` inherits the `eat()` method from the `Animal` class.
- `Dog` also adds its own method `bark()`, which is not available in the `Animal` class.

▼ Some Key Concepts

▼ ✅ Concept 1: Initialize superclass variables in superclass, and subclass variables in subclass

In Java, each class should take care of initializing its own variables. The **subclass constructor** should use `super(...)` to call the **superclass constructor**, ensuring proper and clean initialization.

🔷 Incorrect Approach (Not Recommended)

```
public class Box {
    double l, h, w;

    Box() {
        System.out.println("Box default constructor called...");
        this.l = 0;
        this.h = 0;
        this.w = 0;
    }

    Box(double l, double h, double w) {
        this.l = l;
        this.h = h;
        this.w = w;
    }
}

public class BoxWeight extends Box {
    double weight;

    public BoxWeight(double l, double h, double w, double weight) {
        this.l = l;        // directly accessing superclass variables
        this.h = h;
        this.w = w;
        this.weight = weight;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        BoxWeight box = new BoxWeight(3.0, 4.0, 5.0, 20.0);
        System.out.println("Length: " + box.l);
        System.out.println("Width: " + box.w);
        System.out.println("Height: " + box.h);
        System.out.println("Weight: " + box.weight);
    }
}
```

🔶 Output:

```
Box default constructor called...
Length: 3.0
Width: 5.0
Height: 4.0
Weight: 20.0
```

> ❗ Why this is bad:
>
> When a subclass constructor is called, it *automatically* invokes the superclass constructor first. If not explicitly defined, the default constructor is used, which initializes variables to default values.
>
> Later, those same variables are **overwritten** in the subclass—this is redundant and inefficient.

✅ Correct Approach (Recommended)

```
public class Box {
    double l, h, w;

    Box(double l, double h, double w) {
        this.l = l;
        this.h = h;
        this.w = w;
    }
}

public class BoxWeight extends Box {
    double weight;

    public BoxWeight(double l, double h, double w, double weight) {
        super(l, h, w);          // superclass variables initialized properly
        this.weight = weight;   // subclass variable initialized separately
    }
}
```

🔚 Conclusion:

In Java, each class is responsible for initializing its own variables. The subclass should use `super(...)` in the subclass constructor to delegate initialization of superclass variables to the superclass. Then, initialize the subclass's own variables separately within the subclass constructor. This ensures clean, maintainable, and efficient code. ✅

▼ ✅ Concept 2: What happens if a superclass property is declared `private`? Can it be accessed from a subclass?

In Java, `private` **members of a superclass are not accessible directly** in the subclass — not even through `super.variableName`. This is because `private` members are strictly confined to the class in which they are declared.

So, if a variable like `l` in `Box` is private:

```
public class Box {
    private double l, h, w;

    Box(double l, double h, double w) {
        this.l = l;
        this.h = h;
        this.w = w;
    }
}
```

You **cannot** do this in a subclass:

```
public class BoxWeight extends Box {
    double weight;

    public BoxWeight(double l, double h, double w, double weight) {
        super(l, h, w);
//        this.l = l; ❌ Not allowed because `l` is private in Box
        this.weight = weight;
    }
}
```

✅ How to work with private superclass variables in subclasses?

Use **getters and setters** or pass values via constructor using `super(...)`.

🔷 Example using constructor and getter:

```
public class Box {
    private double l, h, w;
    Box(double l, double h, double w) {
        this.l = l;
        this.h = h;
        this.w = w;
    }

    public double getL() {
        return l;
    }
}

public class BoxWeight extends Box {
    double weight;

    public BoxWeight(double l, double h, double w, double weight) {
        super(l, h, w);
        this.weight = weight;
    }

    public void printLength() {
        System.out.println("Length from Box: " + getL()); // ✅ Access through getter
    }
}
```

✅ **Conclusion**: If a superclass member is `private`, access it using public/protected getters or pass data through constructors. You **cannot** access it directly in the subclass.

▼ ✅ Concept 3: Understanding object creation with superclass/subclass references

You're creating four objects:

```
Box box = new Box();
Box box = new BoxWeight();
BoxWeight box = new BoxWeight();
BoxWeight box = new Box();  // ❌
```

Let's analyze each:

✅ 1. `Box box = new Box();`

This is a **normal object creation** — a `Box` reference pointing to a `Box` object.

```
Box box = new Box(); // calls Box() constructor
```

No issues — full access to all public and protected members of `Box`.

✅ 2. `Box box = new BoxWeight();`

This is **upcasting**. A superclass (`Box`) reference points to a subclass (`BoxWeight`) object.

```
Box box = new BoxWeight(2, 3, 4, 5); // calls BoxWeight constructor, which calls super()
```

```
System.out.println(box.w);      // ✅ allowed
System.out.println(box.weight); // ❌ compile error
```

> **It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed.**

> **When a reference to a subclass object is assigned to a superclass reference variable, you will have access only**

> **to those parts of the object defined by the superclass.**

So you can **only access members defined in `Box`**, not the ones in `BoxWeight` (e.g., `weight`), even though the object is actually a `BoxWeight`.

✅ 3. `BoxWeight box = new BoxWeight();`

This is a **direct subclass object creation** with a `BoxWeight` reference pointing to a `BoxWeight` object.

```
BoxWeight box = new BoxWeight(); // calls BoxWeight(), which may call super()
```

✅ You have full access to both `Box` and `BoxWeight` members.

❌ 4. `BoxWeight box = new Box();`

This is invalid — a **downcast attempt without actual subclass object**.

```
BoxWeight box = new Box(2, 3, 4); // ❌ compile-time error
```

- This line is **trying to assign a `Box` object to a `BoxWeight` reference**.
- Even though `BoxWeight` **is a subclass of `Box`**, this doesn't mean the reverse assignment is allowed. Because you cannot assign a superclass object to a subclass reference **without an explicit cast** — and even with a cast, it will throw `ClassCastException` at runtime.
- This is called **downcasting**, and in this case:
    - The object is of type `Box`.
    - The reference is of type `BoxWeight`.
- **Result: ❌ Compile-time error** or if you use a cast:

    ```
    BoxWeight box = (BoxWeight) new Box(2, 3, 4); // ❌ Compiles but throws ClassCastException at runtime
    ```

Why? Because the object in memory is not actually a `BoxWeight`. It doesn't have the extra `weight` field or any `BoxWeight` methods.

✅ Solution: Upcast First, Then Downcast (Safely)

- **Step 1: Create a `BoxWeight` object (subclass object)**

  ```
  Box box = new BoxWeight(2, 3, 4, 5); // ✅ Upcasting — allowed
  ```

  - Here, a `BoxWeight` object is created, but stored in a `Box` reference.
  - This is allowed because a `BoxWeight` **is-a** `Box`.

- **Step 2: Downcast back to `BoxWeight` (if needed)**

  ```
  if (box instanceof BoxWeight) {
      BoxWeight boxWeight = (BoxWeight) box;  // ✅ Safe Downcasting
      boxWeight.display();                    // ✅ You can now access subclass features
  }
  ```

✅ Full Example:

```
class Box {
    double length, breadth, height;

    Box(double l, double b, double h) {
        length = l;
        breadth = b;
        height = h;
    }
}

class BoxWeight extends Box {
    double weight;
```

```
    BoxWeight(double l, double b, double h, double w) {
        super(l, b, h);
        weight = w;
    }

    void display() {
        System.out.println("Volume: " + (length * breadth * height));
        System.out.println("Weight: " + weight);
    }
}

public class Main {
    public static void main(String[] args) {
        // ✅ Upcasting — subclass object stored in superclass reference
        Box box = new BoxWeight(2, 3, 4, 5);

        // ✅ Safe Downcasting
        if (box instanceof BoxWeight) {
            BoxWeight bw = (BoxWeight) box;
            bw.display();
        }
    }
}
```

🧠 Summary Table:

Statement	Valid?	Explanation
`Box box = new Box();`	✅	Normal instantiation
`Box box = new BoxWeight();`	✅	Upcasting (access only Box members)
`BoxWeight box = new BoxWeight();`	✅	Normal subclass instantiation
`BoxWeight box = new Box();`	❌	Invalid downcast – superclass can't become subclass

▼ ✅ Concept 4: `this.variable` vs `super.variable`

When a subclass **declares a variable or method with the same name** as one in its superclass, the subclass version **hides** the superclass version. In such cases:

- `this.variable` refers to the **subclass's version**.
- `super.variable` refers to the **superclass's version**.

> ✅ Even if the variable names are the same, you can still access the superclass one using super.

🔷 Example:

Let's modify `Box` and `BoxWeight` to both have a variable named `weight` (just for demonstration):

```
public class Box {
    double weight = 10;

    void displayWeight() {
        System.out.println("Box weight: " + weight);
    }
}

public class BoxWeight extends Box {
    double weight = 50;

    void displayWeight() {
        System.out.println("Subclass (BoxWeight) weight: " + this.weight);
        System.out.println("Superclass (Box) weight: " + super.weight);
    }
}
```

🔶 Output:

```
Subclass (BoxWeight) weight: 50.0
Superclass (Box) weight: 10.0
```

✅ This clearly shows how `this.weight` and `super.weight` differ.

> ⚠️ Note: If the superclass's variable is declared private, even super.variable cannot access it.

🧠 When to Use?

- Use `this.variable` when you want to refer to the **current class's** version (especially when local variable names shadow class variables).
- Use `super.variable` to **access hidden members** from the superclass — commonly seen in overridden methods or constructors.

▼ ✅ Concept 5: Superclass is always initialized before subclass using `super()`

In Java, when you create an object of a subclass, the **constructor of the superclass is called first** — either **explicitly using** `super(...)` or **implicitly by default**. Only **after the superclass constructor finishes**, the subclass constructor runs.

> This ensures that all members from the superclass are fully initialized before the subclass adds its own fields.

🔷 Example:

```
public class Box {
    double l, h, w;

    Box(double l, double h, double w) {
        System.out.println("Box constructor called");
        this.l = l;
        this.h = h;
        this.w = w;
    }
}

public class BoxWeight extends Box {
    double weight;

    BoxWeight(double l, double h, double w, double weight) {
          //  this.weight = weight; // if this put first before super() then it will give an error

        super(l, h, w);  // Superclass constructor is called first
        System.out.println("BoxWeight constructor called");
        this.weight = weight;
    }
}
```

🔶 Output when creating object:

```
BoxWeight box = new BoxWeight(2, 3, 4, 5);
```

```
Box constructor called
BoxWeight constructor called
```

🧠 Why this order matters:

- The subclass depends on the fields and logic already set in the superclass.
- Initialization builds **from top of the hierarchy down**.

✅ You can think of it as a pyramid: the base (`super`) is set first, then the top (`subclass`).

▼ ✅ Concept 6: What happens if `super()` is not used in a subclass constructor?

In Java, if you **don't explicitly call** a superclass constructor using `super(...)` in a subclass constructor, the Java compiler will **automatically insert a call to the default (no-arg) constructor** of the superclass — i.e., `super();`.

🔷 Example:

```
public class Box {
    double l, h, w;

    // Default constructor
    Box() {
        System.out.println("Box default constructor called");
        this.l = -1;
        this.h = -1;
        this.w = -1;
    }
}

public class BoxWeight extends Box {
    double weight;

    // No explicit super() here
    BoxWeight() {
        System.out.println("BoxWeight default constructor called");
        this.weight = -1;
    }
}
```

🔶 Output:

```
BoxWeight box = new BoxWeight();
```

```
Box default constructor called
BoxWeight default constructor called
```

⚠️ Important Notes:

1. If the **superclass has no default constructor**, and you don't explicitly call `super(...)`, it will result in a **compile-time error**.
2. That's why, when your superclass only has parameterized constructors, you **must** call `super(...)` manually from the subclass.

🧠 Conclusion:

- ✅ `super()` is **implicitly inserted** only if you **don't write any** **`super(...)`**, and the superclass has a **no-arg constructor**.
- ❌ If superclass only has parameterized constructors, Java **won't guess** which one to call — **you must write it**.

▼ ✅ Concept 7: Passing a subclass object to a superclass constructor

In Java, you can **pass a subclass object** as an argument to a **superclass constructor** if that constructor accepts a parameter of the superclass type.

Even though the object is of the subclass, **Java uses polymorphism**, and since a subclass *is-a* superclass (via inheritance), it's perfectly valid.

🔷 Example from your code:

```
public class Box {
    double l, h, w;

    // Copy constructor
    Box(Box other) {
        System.out.println("Box copy constructor");
        this.l = other.l;
        this.h = other.h;
        this.w = other.w;
    }
}

public class BoxWeight extends Box {
    double weight;

    // Constructor that accepts a BoxWeight object
    BoxWeight(BoxWeight other) {
        super(other); // Passes BoxWeight object to Box(Box other)
        this.weight = other.weight;
    }
}
```

🔶 Output:

```
BoxWeight b1 = new BoxWeight(2, 3, 4, 5);
BoxWeight b2 = new BoxWeight(b1);
```

```
Box class constructor      // from first object creation
Box copy constructor       // from super(other)
```

Even though `other` is a `BoxWeight`, the constructor `Box(Box other)` only copies Box-related fields (`l`, `h`, `w`). It **doesn't care** that the actual object is a subclass — it only accesses what it *knows* (its own fields).

🧠 Key Insight:

> ✅ A superclass reference can accept a subclass object, but it can only access the members defined in the superclass.

This behavior is a classic example of **polymorphism** and **type compatibility** in Java.

🎯 Summary of All 7 Concepts:

Concept	What You Learned
1	How to inherit using `extends`, and subclass includes superclass members except private
2	`private` members in superclass are **not accessible** in subclass directly
3	Difference between `Box box = new BoxWeight()` vs `BoxWeight box = new Box()`
4	`this.variable` refers to current class; `super.variable` accesses superclass
5	Superclass constructor is always called **before** subclass
6	If `super()` is not written, **default superclass constructor** is called automatically
7	You can pass subclass object to superclass constructor if it's expecting superclass type

Different Types of Inheritance in Java

✅ 1. Single Inheritance

What is Single Inheritance?

Single Inheritance means a **subclass inherits from only one superclass**. It forms a **linear parent-child relationship**.

✅ Example Using Box and BoxWeight

```
// Superclass
public class Box {
    double l, h, w;

    Box(double l, double h, double w) {
        this.l = l;
        this.h = h;
        this.w = w;
    }

    void showDimensions() {
        System.out.println("Length: " + l + ", Height: " + h + ", Width: " +
w);
    }
}

// Subclass
public class BoxWeight extends Box {
    double weight;

    BoxWeight(double l, double h, double w, double weight) {
        super(l, h, w);
        this.weight = weight;
    }

    void showWeight() {
        System.out.println("Weight: " + weight);
    }
}

// Test class
public class Main {
    public static void main(String[] args) {
        BoxWeight box = new BoxWeight(2, 3, 4, 5);
        box.showDimensions();  // from Box
        box.showWeight();      // from BoxWeight
    }
}
```

✅ Output:

```
Length: 2.0, Height: 3.0, Width: 4.0
Weight: 5.0
```

✅ Key Point:

- `BoxWeight` inherits all non-private members of `Box`.
- This is the most **basic form of inheritance**, and Java supports it completely.

✅ 2. Multilevel Inheritance

What is Multilevel Inheritance?

Multilevel inheritance means a class is **derived from a class which is already derived from another class** — forming a **chain of inheritance**.

- Grandparent → Parent → Child
- In our case: `Box` → `BoxWeight` → `BoxPrice`

✅ Example Using Box → BoxWeight → BoxPrice

```
// Superclass
public class Box {
    double l, h, w;

    Box(double l, double h, double w) {
        this.l = l;
        this.h = h;
        this.w = w;
    }

    void showDimensions() {
        System.out.println("Length: " + l + ", Height: " + h + ", Width: " +
w);
    }
```

```
}

// Intermediate subclass
public class BoxWeight extends Box {
    double weight;

    BoxWeight(double l, double h, double w, double weight) {
        super(l, h, w);
        this.weight = weight;
    }

    void showWeight() {
        System.out.println("Weight: " + weight);
    }
}

// Subclass of BoxWeight
public class BoxPrice extends BoxWeight {
    double price;

    BoxPrice(double l, double h, double w, double weight, double price) {
        super(l, h, w, weight);
        this.price = price;
    }

    void showPrice() {
        System.out.println("Price: $" + price);
    }
}

// Test class
public class Main {
    public static void main(String[] args) {
        BoxPrice box = new BoxPrice(2, 3, 4, 5, 100);
        box.showDimensions();  // From Box
        box.showWeight();      // From BoxWeight
        box.showPrice();      // From BoxPrice
    }
}
```

✅ Output:

```
Length: 2.0, Height: 3.0, Width: 4.0
Weight: 5.0
Price: $100.0
```

✅ Key Points:

- `BoxPrice` inherits from `BoxWeight`, which itself inherits from `Box`.
- This shows that constructors in the inheritance chain are called **in order from top to bottom** using `super()`.
- Multilevel inheritance is **fully supported in Java** and is useful for building on top of progressively specialized classes.

▼ ✅ 3. Hierarchical Inheritance

What is Hierarchical Inheritance?

In hierarchical inheritance, **multiple subclasses inherit from a single superclass**.

Think of it like **one parent and many children**.

- `Box` → `BoxWeight`
- `Box` → `BoxColor`
- `Box` → `BoxPrice` (directly)

✅ Example Using Box → BoxWeight, BoxColor

```
// Superclass
public class Box {
    double l, h, w;

    Box(double l, double h, double w) {
        this.l = l;
        this.h = h;
        this.w = w;
    }

    void showDimensions() {
        System.out.println("Length: " + l + ", Height: " + h + ", Width: " +
w);
    }
}

// Subclass 1
public class BoxWeight extends Box {
    double weight;

    BoxWeight(double l, double h, double w, double weight) {
        super(l, h, w);
        this.weight = weight;
    }

    void showWeight() {
        System.out.println("Weight: " + weight);
    }
}

// Subclass 2
public class BoxColor extends Box {
    String color;

    BoxColor(double l, double h, double w, String color) {
        super(l, h, w);
        this.color = color;
    }
    void showColor() {
        System.out.println("Color: " + color);
    }
}

// Test class
public class Main {
    public static void main(String[] args) {
        BoxWeight bw = new BoxWeight(2, 3, 4, 5);
        bw.showDimensions();
        bw.showWeight();

        System.out.println("----");

        BoxColor bc = new BoxColor(1, 2, 3, "Red");
        bc.showDimensions();
        bc.showColor();
    }
}
```

✅ Output:

```
Length: 2.0, Height: 3.0, Width: 4.0
Weight: 5.0
----
Length: 1.0, Height: 2.0, Width: 3.0
Color: Red
```

✅ Key Points:

- All subclasses `BoxWeight` and `BoxColor` share the common properties from `Box`.
- Each subclass extends functionality **independently**.

- Hierarchical inheritance is **fully supported in Java**.
- It's great for when you want to reuse a common base across multiple types.

▼ 🚫 4. Multiple Inheritance in Java

What is Multiple Inheritance?

Multiple inheritance means a class inherits from **more than one superclass**.

✅ Example from other languages (like C++):

```
class A {
  // members of A
};

class B {
  // members of B
};

class C : public A, public B {
  // inherits both A and B
};
```

But in **Java**, this is **NOT ALLOWED** for classes.

❌ Why Multiple Inheritance is Not Supported in Java (via Classes)?

Because it leads to **ambiguity**, especially in case of method overriding — known as the **Diamond Problem**.

🧠 The Diamond Problem Explained

```
class A {
    void message() {
        System.out.println("Message from A");
    }
}

class B {
    void message() {
        System.out.println("Message from B");
    }
}


// This is NOT allowed:
// class C extends A, B {  // ❌ ERROR: Class cannot extend multiple classes
// }
```

🧩 Ambiguity:

If class `C` inherits both `A` and `B`, and both override `message()`, then:

> Which version of message() should A inherit? A's or B's?
>
> This ambiguity is why Java **disallows** multiple inheritance through classes.

✅ Then How Does Java Support Multiple Inheritance?

Through interfaces!

In Java, a class can implement **multiple interfaces**.

```
interface A {
    void show();
}

```

```
interface B {
    void show();
}

class C implements A, B {
    public void show() {
        System.out.println("Hello from C");
    }
}
```

✅ No ambiguity here because **interfaces don’t provide method bodies (unless they’re default/static)** — the class provides its own implementation.

✅ Summary:

Concept	Allowed in Java?	Reason
Multiple class inheritance	❌	Causes ambiguity
Multiple interface inheritance	✅	No ambiguity, handled by implementation

▼ Key Concepts on Different types of Inheritance

✅ 🔑 Key Concepts for Single Inheritance

- **Only one superclass** is extended.
- Subclass inherits non-private members of the superclass.
- Can override superclass methods or add new ones.
- Allows method/variable reuse and specialization.

🔷 Constructor Chaining:

- `super()` is used to call the superclass constructor from the subclass.

- Always the first statement in subclass constructor.

```
BoxWeight(double l, double h, double w, double weight) {
   super(l, h, w); // calls Box constructor
   this.weight = weight;
}
```

✅🔑 Key Concepts for Multilevel Inheritance

- A subclass inherits from another subclass (a chain).
- All constructors in the hierarchy are executed **top to bottom**.
- `super()` propagates the initialization up the chain.

🔷 Example Chain:

`Box` → `BoxWeight` → `BoxPrice`

🔷 Execution Order:

When `new BoxPrice(...)` is created:

1. `Box` constructor is called
2. then `BoxWeight` constructor
3. then `BoxPrice` constructor

🔷 Key Point:

> Each class in the hierarchy should only initialize its own variables.
>
> Use `super()` to let parent class handle its part.

✅🔑 Key Concepts for Hierarchical Inheritance

- One superclass, multiple subclasses.

- Each subclass inherits the **same base properties**, but implements different features.
- Each child class works independently.

🔷 Benefit:

> Great for code reuse and when modeling "is-a" relationships from a common base.

🔷 Practical Use Case:

`Box` → `BoxWeight`, `BoxColor`, `BoxMaterial`

Each subclass adds **a unique feature** to a shared structure.

🚫 🔑 Key Concepts for Multiple Inheritance

- **Not supported** via classes in Java.
- Prevents **Diamond Problem** (method ambiguity).
- Supported via **interfaces** instead.

🔷 Why Not Allowed via Classes:

If two superclasses have the same method signature, the subclass won't know which one to inherit.

🔷 Workaround:

Use **interfaces**:

```
interface Printable { void show(); }
interface Scannable { void show(); }

class Machine implements Printable, Scannable {
    public void show() {
        System.out.println("Machine feature");
    }
}
```