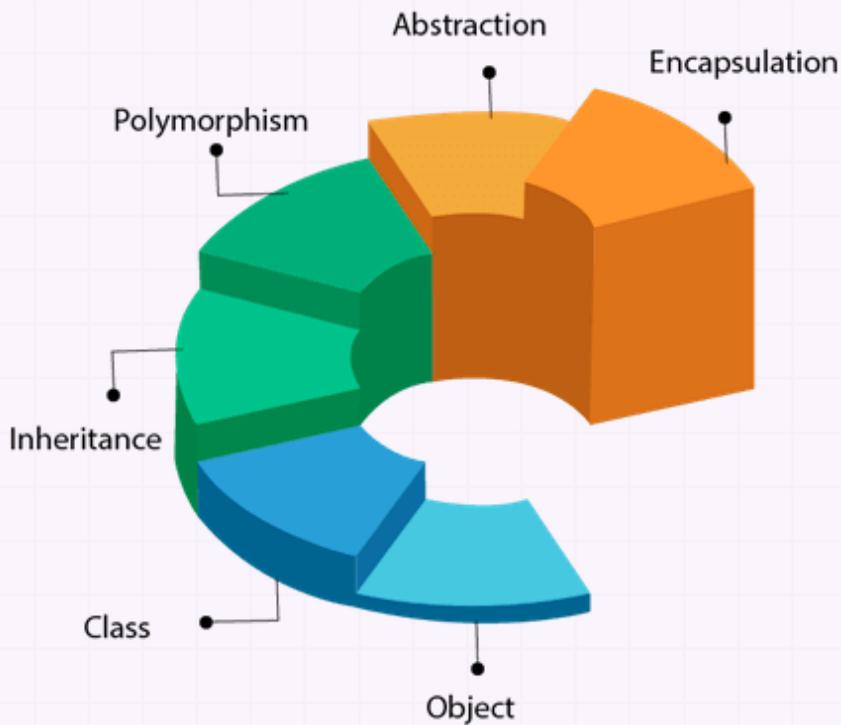




**InterviewCafe**  
Where Learning Never Takes a Break!

# OOPS Concepts with Real-Life Examples in Java



---

**Santosh Kumar Mishra**

Software Engineer at Microsoft • Author • Founder of InterviewCafe



# Introduction

**Object-oriented programming (OOP)** is a powerful paradigm used in Java to structure code in a way that models real-world entities.

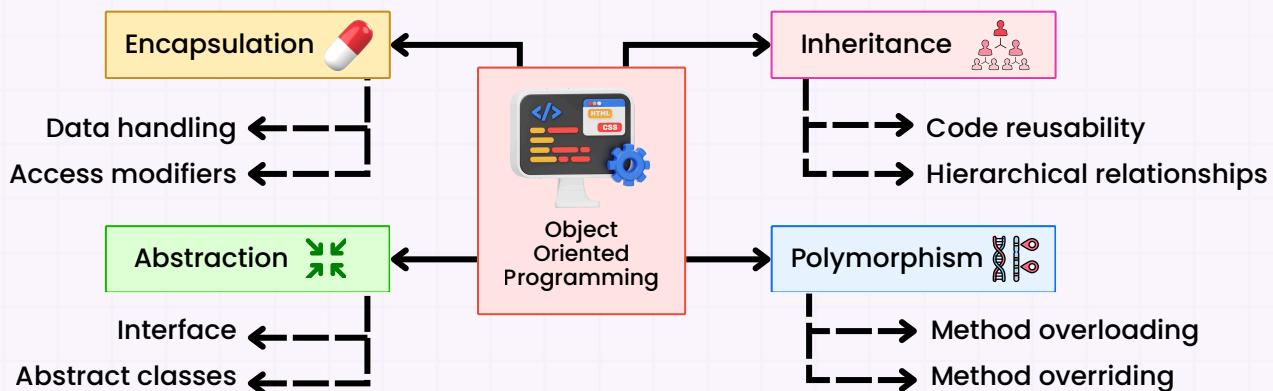
This guide will explore the four key principles of OOP:

1. Encapsulation

2. Inheritance

3. Polymorphism

4. Abstraction



We'll also cover Constructors and Overloading.

For each concept, we'll provide:

- A clear explanation
- Detailed real-life examples
- Practical Java code examples

Let's dive in!

## 1. Encapsulation

- **Encapsulation** refers to the bundling of data (fields) and methods (functions) that operate on the data into a single unit or class.
- It restricts direct access to certain components, protecting the integrity of the data by controlling how it is accessed and modified.



**BUY  
NOW**

**Don't miss out—Unlock the full book  
now and save 25% OFF with code:  
**CRACKJAVA25** (Limited time offer!)**

[\*\*GET NOW\*\*](#)

**Crack Java Interview  
Like Pro**

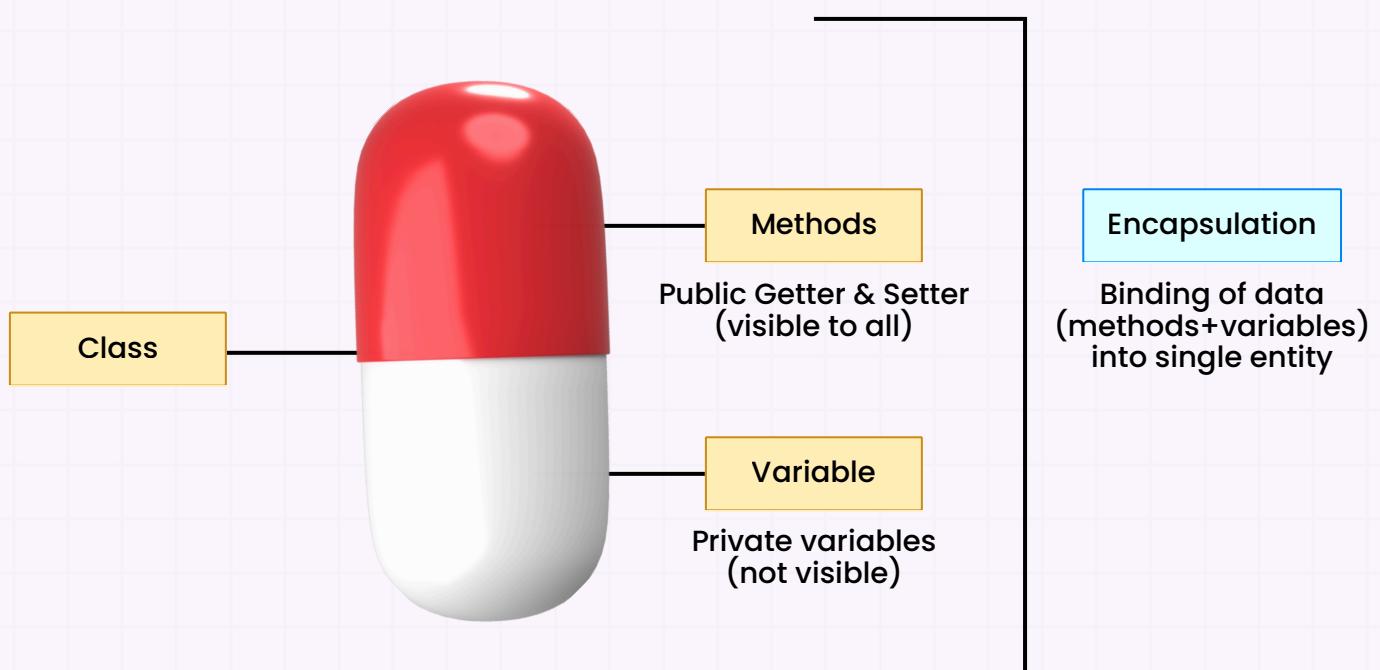


## Real-Life Examples

### Example 1: Medical Capsule

Imagine a medical capsule that contains different medicines mixed together.

- **How it relates to Encapsulation:**
  - The capsule shell represents the class in Java.
  - The medicines inside are like the private data members.
  - The capsule's outer layer (the class) protects and contains the internal components (the data).
  - Users (patients) don't need to know about the individual ingredients or how they're mixed.
  - They only interact with the capsule as a whole, similar to how we interact with public methods in a class.



## Example 2: Large Organization

Think of a large corporation with multiple departments like HR, Finance, and Marketing.

- **How it relates to Encapsulation:**

- The entire organization is like a class in Java.
- Each department is similar to a private data member or method.
- Employees in one department don't directly access or modify data from another department.
- Communication between departments happens through official channels (like public methods in Java).
- The outside world interacts with the company as a single entity, not with individual departments directly.



## Java Example

- In Java Encapsulation is implemented using private variables and providing public getter and setter methods to access and modify these variables.
- This protects the integrity of the data and hides the details of the class implementation.

```
public class BankAccount {  
    private double balance; // Private data  
  
    public BankAccount(double initialBalance) {  
        this.balance = initialBalance;  
    }  
  
    public void deposit(double amount) {  
        if (amount > 0) balance += amount;  
    }  
  
    public double getBalance() {  
        return balance;  
    }  
}
```

### In this example:

- **balance** and **accountNumber** are private variables, meaning they can't be accessed directly from outside the class.
- Public methods like **getBalance()**, **deposit()**, and **withdraw()** provide controlled access to these private variables.
- The internal implementation of how the balance is stored and modified is hidden from the outside world, demonstrating encapsulation.

## 2. Inheritance

- **Inheritance** is the mechanism by which one class (the child or subclass) inherits properties and behaviors from another class (the parent or superclass).
- This enables code reuse and helps maintain hierarchical relationships between classes.

### Mom and Daughter



## Real-Life Examples

### Example 1: Vehicle Hierarchy

Consider different types of vehicles on the road.

- How it relates to Inheritance:
  - "Vehicle" would be the parent class.
  - Cars, bicycles, and buses are child classes that inherit from Vehicle.
  - All vehicles share common properties (like having wheels, a method of propulsion).
- Each specific type of vehicle adds its own unique properties:
  - Cars have doors and a trunk.
  - Bicycles have pedals and handlebars.
  - Buses have multiple rows of seats.

## Vehicles



## Example 2: Animal Kingdom

Think about different animals in nature.

- **How it relates to Inheritance:**
  - "Animal" would be the parent class.
  - Dogs, cats, and horses are child classes that inherit from Animal.
  - All animals share common properties (like needing food, having a lifespan).
- **Each specific animal adds its own unique properties:**
  - Dogs can bark and wag their tails.
  - Cats can purr and retract their claws.
  - Horses can gallop and neigh.



## Animals

## Java Example

- In Java Inheritance is implemented using the **extends** keyword.
- A subclass inherits all non-private members (fields, methods, and nested classes) from its superclass.
- Constructors are not inherited, but the subclass can call the superclass constructor using **super()**.



```
// Superclass
class Vehicle {
    public void start() { System.out.println("Vehicle starting"); }
}

// Subclass
class Car extends Vehicle {
    private int numberOfDoors;

    public Car(String brand, int year, int numberOfDoors) {
        super(brand, year); // Call to superclass constructor
        this.numberOfDoors = numberOfDoors;
    }

    public void honk() { System.out.println("Car honking"); }

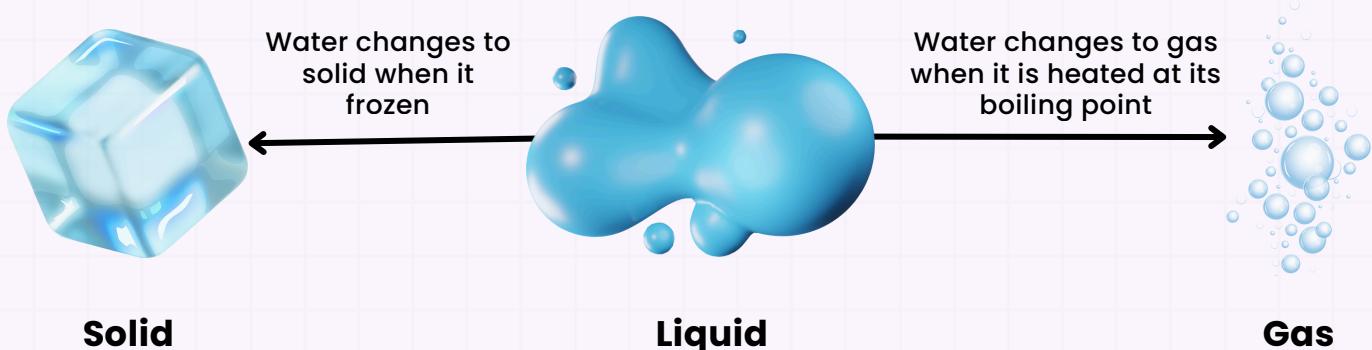
    // Overriding the superclass method
    @Override
    public void start() {
        System.out.println("The car is starting");
    }
}
```

### In this example:

- **Car** is a subclass of **Vehicle**, inheriting its properties and methods.
- **Car** adds its own property (**numberOfDoors**) and method (**honk()**).
- **Car** overrides the **start()** method to provide its own implementation.
- The **super()** call in the **Car** constructor ensures proper initialization of inherited properties.

### 3. Polymorphism

- **Polymorphism** allows one object to take many forms.
- In Java, polymorphism is primarily achieved through method overloading (compile-time polymorphism) and method overriding (runtime polymorphism).



**BUY  
NOW**

**Don't miss out - Unlock the full book now and save 25% OFF with code: CRACKJAVA25 (Limited time offer!)**

[\*\*GET NOW\*\*](#)

**Crack Java Interview Like Pro**

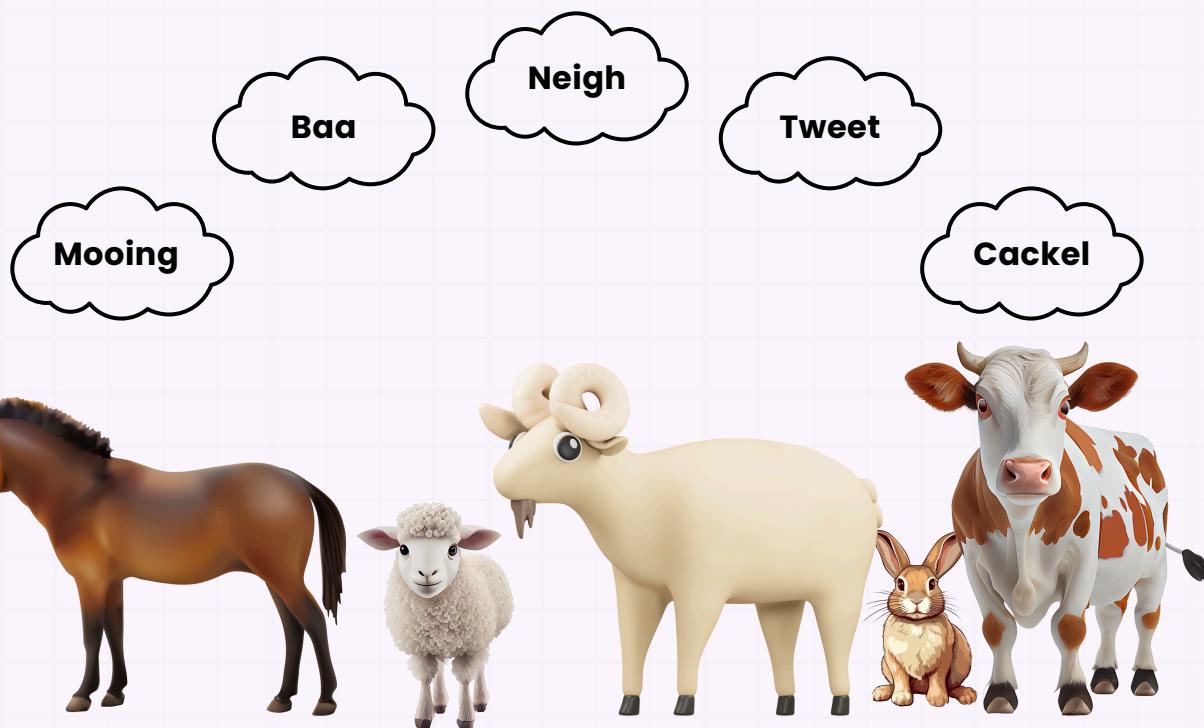


## Real-Life Examples

### Example 1: Sound of Animals

Think about how a single person plays different roles in different contexts.

- **How it relates to Polymorphism:**
  - A person can be a student, a teacher, and a parent.
- **The same individual behaves differently in each role:**
  - As a student, they attend classes and submit assignments.
  - As a teacher, they prepare lessons and grade papers.
  - As a parent, they care for their children and manage household responsibilities.
- This is similar to how an object in Java can be treated as different types depending on the context, exhibiting different behaviour's.

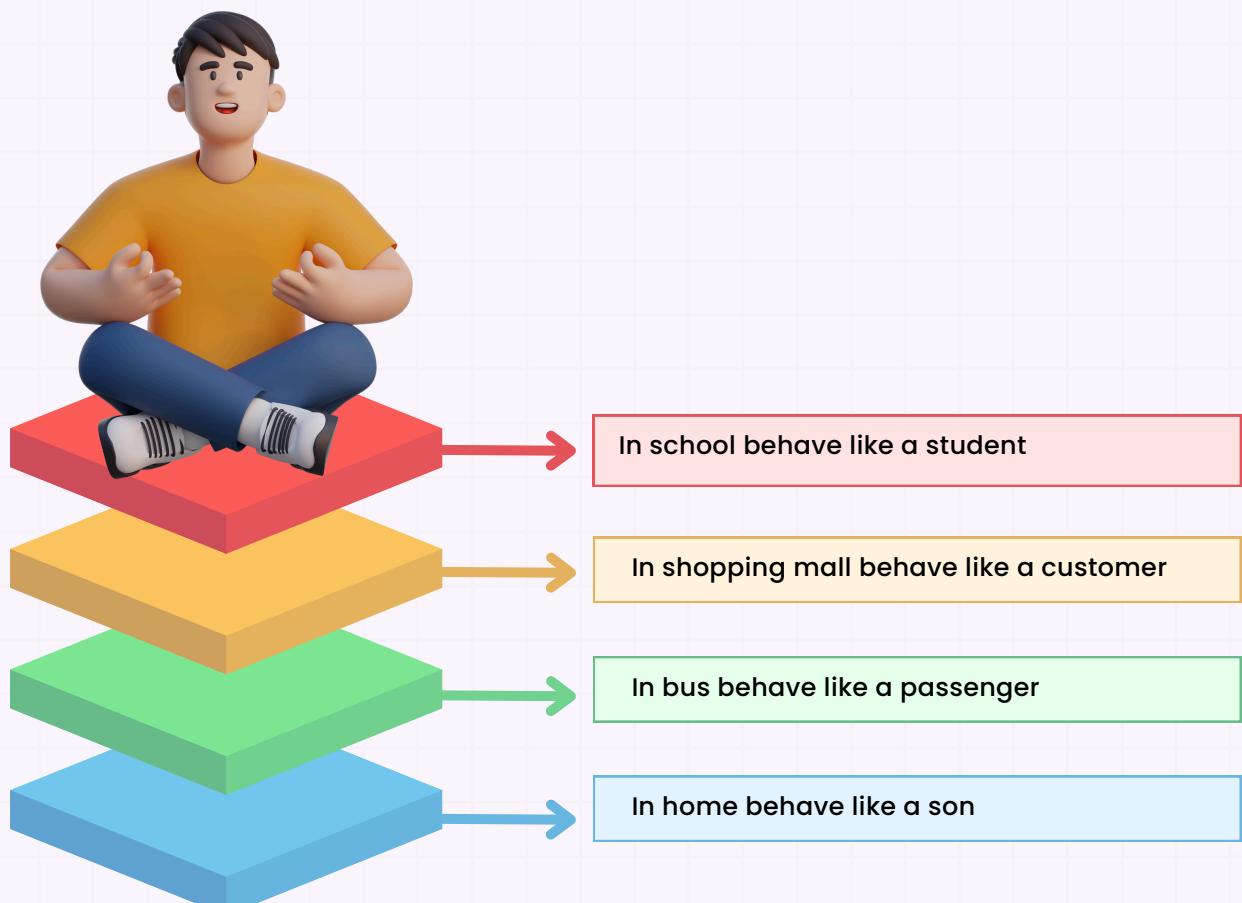


## Sound of Animals

## Example 2: A Person's Roles

Think about different animals in nature.

- **How it relates to Inheritance:**
  - "Animal" would be the parent class.
  - Dogs, cats, and horses are child classes that inherit from Animal.
  - All animals share common properties (like needing food, having a lifespan).
- **Each specific animal adds its own unique properties:**
  - Dogs can bark and wag their tails.
  - Cats can purr and retract their claws.
  - Horses can gallop and neigh.
- This is similar to how an object in Java can be treated as different types depending on the context, exhibiting different behaviour's.



## Java Example

- **Method Overloading** is achieved by defining multiple methods with the same name but different parameters in the same class.
- **Method Overriding** is implemented by providing a new implementation for a method in a subclass that is already defined in its superclass.

```
● ● ●

class Shape {
    public double calculateArea(double radius) { return Math.PI * radius * radius; } // Circle
    public double calculateArea(double length, double width) { return length * width; } // Rectangle
}

class Circle extends Shape {
    private double radius;
    public Circle(double radius) { this.radius = radius; }
    @Override
    public double calculateArea() { return Math.PI * radius * radius; }
}

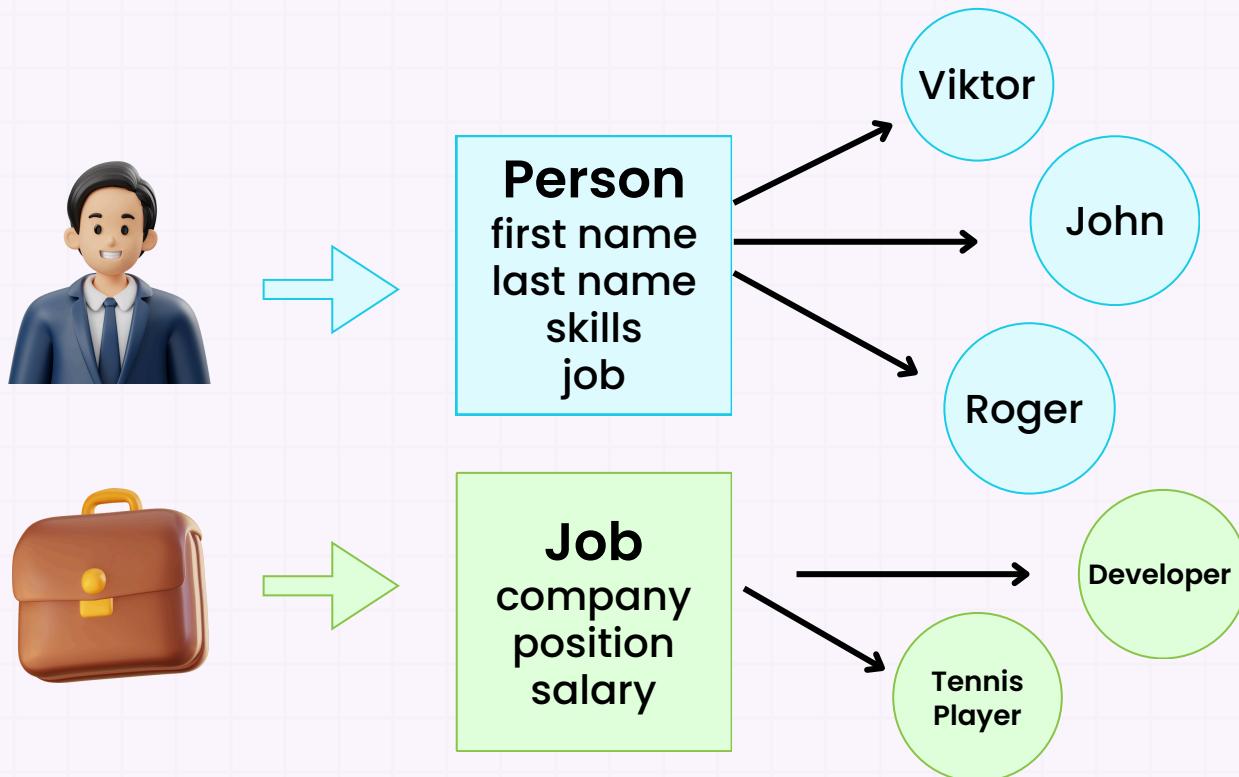
class Rectangle extends Shape {
    private double length, width;
    public Rectangle(double length, double width) { this.length = length; this.width = width; }
    @Override
    public double calculateArea() { return length * width; }
}
```

### In this example:

- **Overloading**: calculateArea in Shape for different parameters.
- **Overriding**: calculateArea in Circle and Rectangle provide specific implementations.

## 4. Abstraction

- **Abstraction** involves hiding the complex internal workings of a system and exposing only what is necessary for the user.
- This helps simplify complex systems by focusing on what the user needs to know.



## Abstraction

**BUY  
NOW**

Don't miss out - Unlock the full book now and save **25% OFF** with code:  
**CRACKJAVA25** (Limited time offer!)

[GET NOW](#)

Crack Java Interview Like Pro

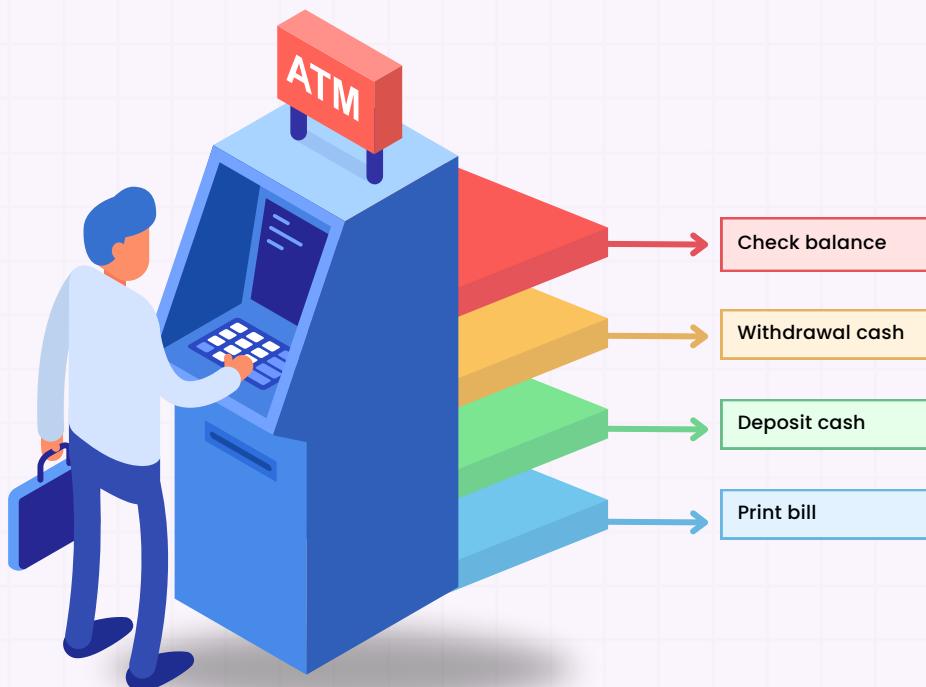


## Real-Life Examples

### Example 1: ATM Machine

Consider how we interact with an ATM machine.

- **How it relates to Abstraction:**
  - Users only see a simple interface: screen, keypad, card slot, and cash dispenser.
- **They don't need to know about the complex internal processes:**
  - How the ATM communicates with the bank's servers.
  - How it verifies account details and processes transactions.
  - How it manages its internal cash supply.
- Users only need to know which buttons to press to perform actions like withdrawing cash or checking their balance.
- This is like how abstract classes or interfaces in Java hide complex implementations behind simple, usable methods.



Even though it performs a lot of actions it doesn't show us the process. It has hidden its process by showing only the main things like getting inputs and giving the output.

## Example 2: Mobile Phone

Think about using a smartphone.

- **How it relates to Abstraction:**
  - Users interact with apps through a touchscreen interface.
- **They don't need to understand:**
  - How the phone processes touch inputs.
  - How it renders graphics on the screen.
  - How it manages memory or processes data.
- Users simply tap icons to open apps, swipe to navigate, or pinch to zoom.
- This abstraction of complex processes behind a simple interface is similar to how Java interfaces or abstract classes work, providing a simple way to interact with complex systems.



- In Java Abstraction is implemented using abstract classes and interfaces.
- Abstract classes can have both abstract and concrete methods, while interfaces (prior to Java 8) only have abstract methods.

```
● ● ●

abstract class Device {
    public abstract void performFunction();
}

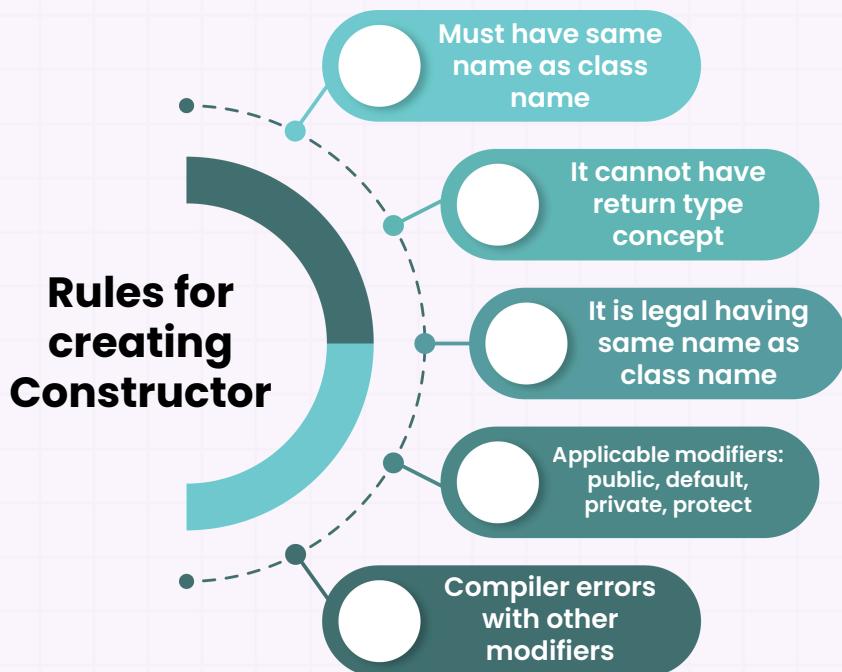
class Television extends Device {
    public void performFunction() { System.out.println("Displaying content"); }
}
```

### In this example:

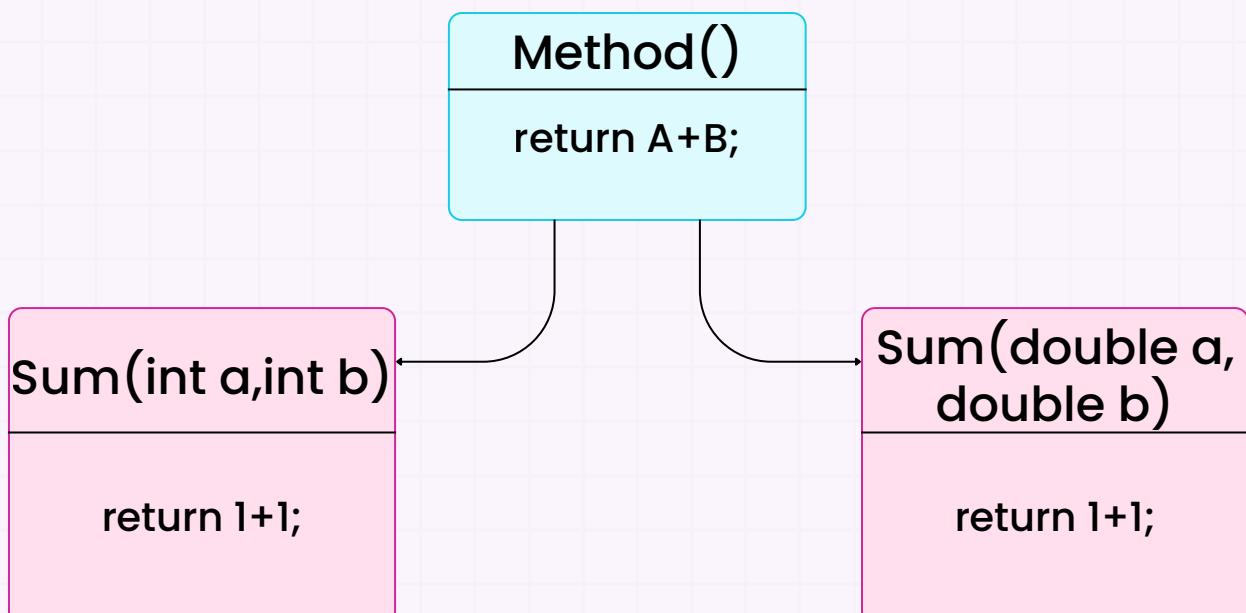
- **ElectronicDevice** is an abstract class with both abstract (**performFunction()**) and concrete (**turnOn()**, **turnOff()**) methods.
- **Television** is a concrete class that extends **ElectronicDevice** and provides an implementation for the abstract method.
- **RemoteControlled** is an interface defining additional behaviors.
- **SmartTV** demonstrates multiple inheritance by extending **ElectronicDevice** and implementing **RemoteControlled**.

## 5. Constructors and Overloading

- A **constructor** is a special method used to initialize objects.



- Constructor overloading** allows a class to have more than one constructor with different parameter lists.



## Overloading in Java

## Real-Life Examples

### Example: Pizza Ordering

Imagine ordering a pizza at a restaurant.

- How it relates to Constructors and Overloading:
  - The basic pizza (just crust and cheese) is like a default constructor.
- Adding toppings is similar to using overloaded constructors:
  - You can order a pizza with just one topping (like a constructor with one parameter).
  - Or you can order a pizza with multiple toppings (like a constructor with multiple parameters).
- The pizza remains fundamentally the same (it's still a pizza), but its initialization (how it's made) varies based on your order.
- This is analogous to how different constructors in a Java class can initialize an object in different ways, but they all create an instance of the same class.



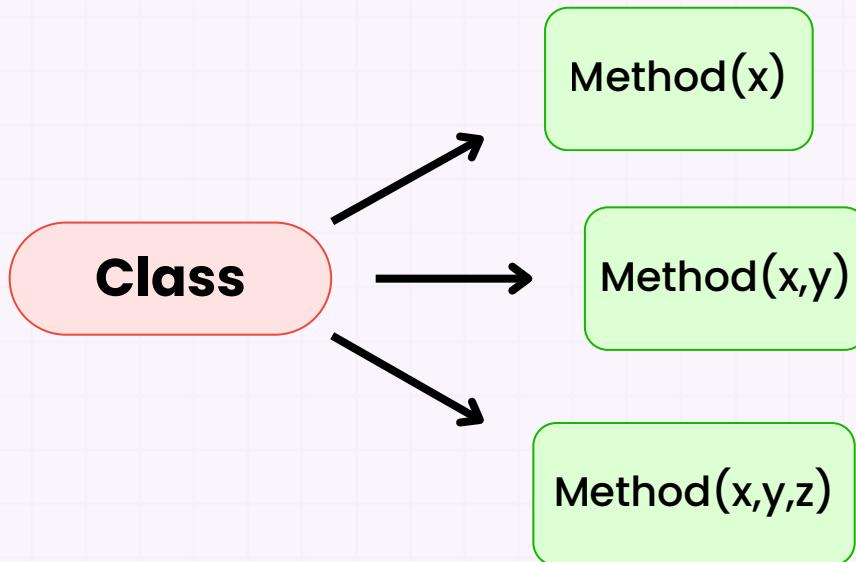
## Java Example

This example shows how constructor overloading allows for flexible object creation.

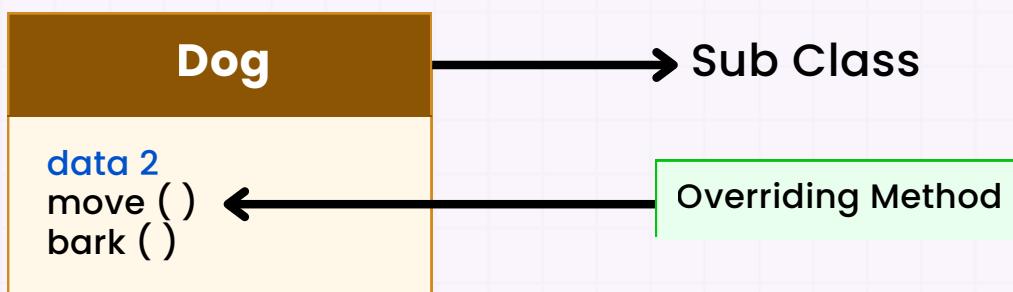
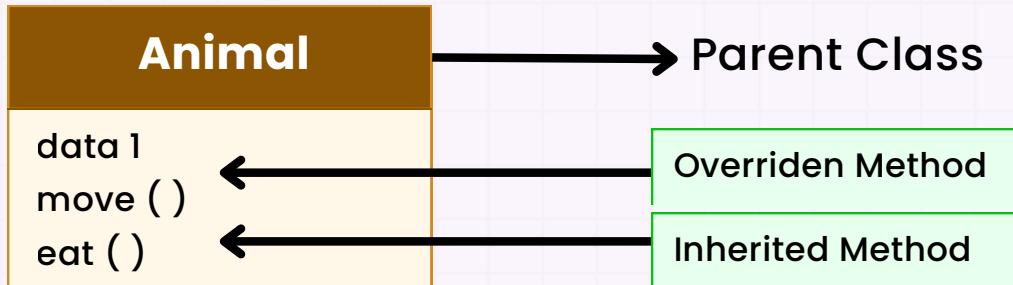
```
● ● ●  
public class Pizza {  
    public Pizza() { System.out.println("Plain pizza"); }  
    public Pizza(String... toppings) { System.out.println("Pizza with " + String.join(", ", toppings)); }  
}
```

## 6. Method Overloading vs. Overriding

- Method Overloading is when multiple methods have the same name but different parameter lists.
- It occurs within the same class and is resolved at compile-time.



- Method Overriding is when a subclass provides a specific implementation of a method that is already defined in the parent class.
- It occurs at runtime (runtime polymorphism).

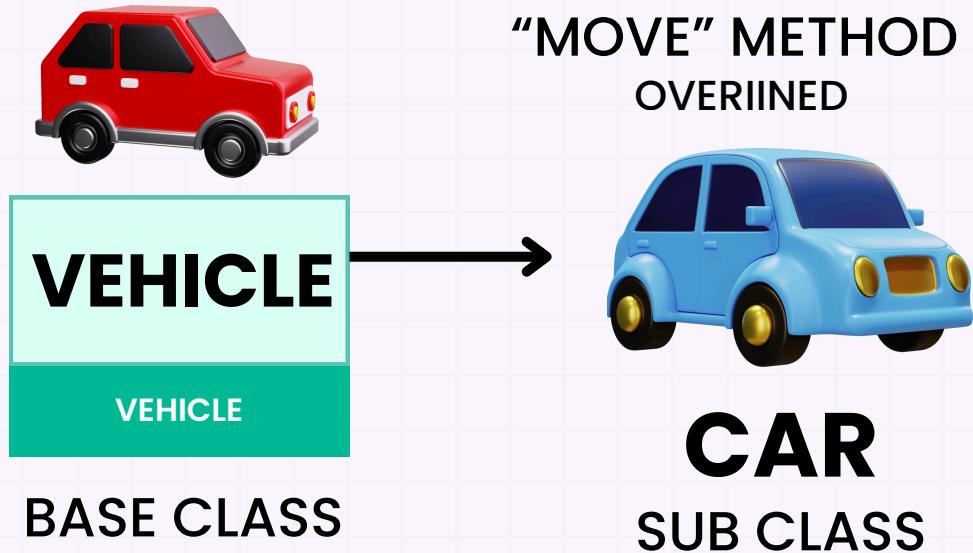


## Real-Life Examples

- **Overloading:** Think of a printer that can print either text or images.
- Even though both methods are called "**print**," they handle different types of data.



- **Overriding:** Imagine you have a base class Vehicle that moves.
- A Car (subclass) might override this method to provide its specific way of moving.



## Java Example

```
// Overloading
class Printer {
    // Overloading
    void print(String text) {
        System.out.println("Printing text: " + text);
    }

    void print(int number) {
        System.out.println("Printing number: " + number);
    }
}

// Overriding
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}
```

# Crack Java Interview Like Pro

500+ Java Interview Questions (with Answers)

Visual Diagrams + Real-Life Examples

Java-Focused System Design

Resume & Shortlisting Strategies

300+ Chapter-wise Quizzes

Mini Projects in Every Chapter

Behavioral & Situational Round Prep

30+ Major Projects + 50+ Mini Project Ideas

Grab 25% Flat Discount!

**BUY NOW**