# 🌟 What is Polymorphism in Java?

**Polymorphism** means "**many forms**".  
In Java, it allows **one action to behave differently** depending on the object performing it.

👉 It lets the **same method name** perform **different tasks**.

**🧠 Why is Polymorphism Important?**

* Makes code **reusable** and **flexible**
* Supports **extensibility** — new behavior can be added without changing existing code
* Enables **method overriding** and **dynamic method dispatch**

**🧩 Types of Polymorphism in Java**

| **Type** | **When It Happens** | **How It's Achieved** |
| --- | --- | --- |
| **Compile-time Polymorphism** | At compile time | **Method Overloading** |
| **Runtime Polymorphism** | At runtime | **Method Overriding** |

**🔷 1. Compile-Time Polymorphism (Method Overloading)**

**🧾 Definition:**

Same method name, **different parameters** (type, number, or order) in same class

**✅ Real-World Analogy:**

Imagine a printer:

* print(String document)
* print(Image image)
* print(PDF pdf, int copies)

All are named print, but they handle different types of content.

👨‍💻 Example 1:  
class Calculator {

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

int add(int a, int b, int c) {

return a + b + c;

}

}

public class Test {

public static void main(String[] args) {

Calculator calc = new Calculator();

System.out.println(calc.add(10, 20)); // int

System.out.println(calc.add(5.5, 3.3)); // double

System.out.println(calc.add(1, 2, 3)); // 3-arg method

}

}  
  
👨‍💻 Example 2: **cook() in a Restaurant App**  
class Chef {

void cook(String dish) {

System.out.println("Cooking: " + dish);

}

void cook(String dish, int quantity) {

System.out.println("Cooking " + quantity + " plates of " + dish);

}

void cook(String dish, boolean isSpicy) {

String spice = isSpicy ? "with spice" : "without spice";

System.out.println("Cooking " + dish + " " + spice);

}

}  
🧠 Use Case: In a restaurant app, the same cook method is used to process different types of orders — default, with quantity, or with special preferences.  
  
👨‍💻 Example 3: sendMessage() in a Chat App  
class Messenger {

void sendMessage(String msg) {

System.out.println("Sending message: " + msg);

}

void sendMessage(String msg, String userId) {

System.out.println("Sending to user " + userId + ": " + msg);

}

void sendMessage(String msg, String[] userIds) {

System.out.println("Broadcasting: " + msg + " to multiple users.");

}

}  
**🧠 Use Case:** In a **messaging app** like WhatsApp, sending messages to individuals or groups can use overloaded methods with the same name.  
  
  
In Java, you cannot overload methods based solely on their return type. Method overloading depends on having different method signatures, which are determined by:

1. **Method name**
2. **Parameter types** (number, order, and types of parameters)
3. **Not considered**: Return type, access modifiers (public, private, etc.), or exceptions.

**❌ Invalid Overloading (Same Parameters, Different Return Types)**public class Example {

public int getValue() { return 1; }

public String getValue() { return "hello"; } // ❌ Compile-time error!

}  
This will fail because the compiler cannot distinguish which method to call based only on return type.  
  
**✔️ Valid Overloading (Different Parameters)**

public class Example {

public int getValue(int x) { return x; }

public String getValue(String s) { return s; } // ✅ Valid (different parameter types)

}  
**Why Java Doesn’t Allow Return-Type-Based Overloading?**

* **Ambiguity in method calls**: The compiler wouldn’t know which method to invoke if the return type were the only difference.

getValue(); // Should it call the int or String version? Compiler can't decide!

Workaround:  
If you need different return types, either:

1. **Use different method names:**

public int getIntValue() { return 1; }

public String getStringValue() { return "hello"; }  
  **2. Use generics (for more advanced cases):**  
public <T> T getValue(Class<T> type) {

if (type == Integer.class) return (T) Integer.valueOf(1);

else if (type == String.class) return (T) "hello";

else return null;

}  
  
**Key Takeaway:**

Java method overloading requires differences in **method parameters**, not just the return type.

🔷 2. Runtime Polymorphism (Method Overriding)  
**🧾 Definition:**

Same method name and parameters, but **defined in parent and child classes**.  
At runtime, the **child class's version** is executed.

**✅ Real-World Analogy:**

A **generic remote** can be used for many TVs, but each TV **overrides** the turnOn() behavior differently.

👨‍💻 Example1:  
class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

void sound() {

System.out.println("Dog barks");

}

}

public class Test {

public static void main(String[] args) {

Animal a = new Dog(); // Parent reference, child object

a.sound(); // Runtime decides which version to call

}

}  
  
**👨‍💻 Example 2: start() in a Vehicle System**  
class Vehicle {

void start() {

System.out.println("Starting generic vehicle...");

}

}

class Bike extends Vehicle {

void start() {

System.out.println("Bike started with a kick");

}

}

class Car extends Vehicle {

void start() {

System.out.println("Car started with a key");

}

}  
public class Test {

public static void main(String[] args) {

Vehicle v1 = new Bike();

Vehicle v2 = new Car();

v1.start(); // Output: Bike started with a kick

v2.start(); // Output: Car started with a key

}

}  
🧠 Use Case: In a transportation management system, you override the start() method based on the vehicle type.  
  
**👨‍💻 Example 3: calculateCharges() in an E-Commerce App**class Payment {

void calculateCharges() {

System.out.println("Basic processing fee applied.");

}

}

class CreditCardPayment extends Payment {

void calculateCharges() {

System.out.println("2% credit card charge applied.");

}

}

class PayPalPayment extends Payment {

void calculateCharges() {

System.out.println("1.5% PayPal fee applied.");

}

}🧠 Use Case: In an e-commerce app, different payment methods override how charges are applied, while maintaining a common method name.  
  
In Java, **method overriding** depends on the **runtime type of the object (object type)**, not the **reference type**. This is a fundamental aspect of **polymorphism** in Java.

**Key Points:**

1. **Reference Type** → Determines **what methods can be called** (compile-time check).
2. **Object Type (Runtime Type)** → Determines **which overridden method is executed** (runtime behavior).

class Animal {

void makeSound() {

System.out.println("Animal sound");

}

}

class Dog extends Animal {

@Override

void makeSound() {

System.out.println("Bark");

}

}

public class Main {

public static void main(String[] args) {

Animal myAnimal = new Dog(); // Reference type = Animal, Object type = Dog

myAnimal.makeSound(); // Outputs "Bark" (Dog's method is called)

}

}

* **Reference type (Animal)** → Only allows calling methods defined in Animal.
* **Object type (Dog)** → At runtime, Java uses **dynamic method dispatch** to call the overridden method in Dog.

**What Happens If the Method is Not Overridden?**

If the subclass **does not override** the method, the **parent class's method** is called:

class Cat extends Animal {

// No makeSound() override

}

public class Main {

public static void main(String[] args) {

Animal myCat = new Cat();

myCat.makeSound(); // Outputs "Animal sound" (inherited from Animal)

}

}

**Important Rules:**

1. **Overriding depends on the actual object type (runtime type)**.
2. **The reference type only restricts which methods can be called** (compile-time check).
3. **@Override annotation** helps ensure correct overriding (optional but recommended).

**Special Cases:**

* **Static methods** → **Not overridden**, they follow **reference type** (compile-time binding).
* **Private methods** → **Cannot be overridden** (not visible to subclasses).
* **Final methods** → **Cannot be overridden**.

**Summary:**

| **Scenario** | **Method Called** |
| --- | --- |
| **Overridden method** | **Runtime type (object type)** decides |
| **Non-overridden method** | **Parent class method** executes |
| **Static method** | **Reference type** decides (no overriding) |

This behavior is why Java supports polymorphism—allowing flexible and dynamic method resolution at runtime.  
  
🔍 Comparison at a Glance

| **Feature** | **Method Overloading** | **Method Overriding** |
| --- | --- | --- |
| Polymorphism Type | Compile-time | Runtime |
| Class Structure | Same class | Parent-child classes |
| Signature | Must differ (params) | Must be same (params + return type) |
| Flexibility | Provides multiple ways to use a method | Enables different behavior in subclasses |

**💡 Where Do Real-World Applications Use These?**

| **Domain / App** | **Overloading Used For** | **Overriding Used For** |
| --- | --- | --- |
| **Banking** | **withdraw(amount), withdraw(amount, atm)** | **calculateInterest() for different account types** |
| **Ride-Sharing App** | **bookRide(location), bookRide(location, carType)** | **calculateFare() for Auto, Car, Bike** |
| **E-commerce** | **addToCart(item), addToCart(item, qty)** | **applyDiscount() overridden for Sale, Coupon** |
| **Hospital Management** | **registerPatient(name), registerPatient(name, age)** | **treatPatient() overridden for departments** |
| **Gaming** | **attack() with different weapons** | **move() behavior overridden by Player types** |

**🎮 Guess the Output Exercises  
🔹 Exercise 1: Method Overloading**class PrintService {

void print(String data) {

System.out.println("Printing String: " + data);

}

void print(int data) {

System.out.println("Printing Integer: " + data);

}

}

public class Test {

public static void main(String[] args) {

PrintService p = new PrintService();

p.print("Hello");

p.print(123);

}

}  
🔹 Exercise 2: Method Overriding  
class Vehicle {

void start() {

System.out.println("Vehicle is starting");

}

}

class Car extends Vehicle {

void start() {

System.out.println("Car is starting");

}

}

public class Test {

public static void main(String[] args) {

Vehicle v = new Car();

v.start();

}

}  
  
🔹 Exercise 3: Polymorphism + Inheritance

class Parent {

void greet() {

System.out.println("Hello from Parent");

}

}

class Child extends Parent {

void greet() {

System.out.println("Hello from Child");

}

}

public class Test {

public static void main(String[] args) {

Parent obj1 = new Parent();

Parent obj2 = new Child();

obj1.greet();

obj2.greet();

}

}  
  
🔹 Exercise 4: Method Overloading  
class Display {

void show(int a) {

System.out.println("Integer: " + a);

}

void show(String a) {

System.out.println("String: " + a);

}

}

public class Main {

public static void main(String[] args) {

Display d = new Display();

d.show(10);

d.show("Java");

}

}  
  
🔹 Exercise 5: Method Overloading  
class Parent {

void greet() {

System.out.println("Hello from Parent");

}

}

class Child extends Parent {

void greet() {

System.out.println("Hello from Child");

}

}

public class Test {

public static void main(String[] args) {

Parent p = new Child();

p.greet();

}

}

🔁 **Polymorphism in the JDK: Real Examples**✅ 1. **Collection Framework – Method Overriding**List<String> list = new ArrayList<>();

list.add("Java");

list.add("Python");  
  
- List is an **interface**.

- ArrayList is a **class** that implements List.

- When you write List<String> list = new ArrayList<>();, you're using **runtime polymorphism**.  
  
✅ 2. **Method Overloading in PrintStream (System.out)**System.out.println("Hello World");

System.out.println(42);

System.out.println(3.14);

1] All these are calls to println(), but with different types.

2] The PrintStream class **overloads** the println() method for:

* String, int, double, char[], boolean, Object, etc.

public void println(String x) { ... }

public void println(int x) { ... }

public void println(double x) { ... }

## 🛡️ Encapsulation in Java

**✅ Definition:**

Encapsulation is the **binding of data (variables)** and **methods (functions)** that operate on the data into a **single unit** (class). It hides the internal state of the object from the outside world and allows access only through **public methods**.

**🔒 Why Encapsulation?**

* Protects data from unauthorized access or modification
* Improves code maintainability
* Supports abstraction by hiding complex details
* Enables data validation and security

**🏠 Real-World Analogy:**

Think of a **bank ATM**:

* You insert your card and enter your PIN (public access methods).
* The ATM does internal processing — **you don’t see the backend logic** (data is private).
* You can't directly change your account balance; only approved methods can do that (controlled access)

🧑‍💻 **Java Code Example:**class BankAccount {

private String accountHolder;

private double balance;

// Constructor

public BankAccount(String name, double initialBalance) {

this.accountHolder = name;

this.balance = initialBalance;

}

// Public getter

public double getBalance() {

return balance;

}

// Public setter (with validation)

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

System.out.println("Deposited: $" + amount);

} else {

System.out.println("Invalid deposit amount.");

}

}

// Withdraw method

public void withdraw(double amount) {

if (amount > 0 && amount <= balance) {

balance -= amount;

System.out.println("Withdrawn: $" + amount);

} else {

System.out.println("Invalid withdrawal attempt.");

}

}

}  
  
public class Main {

public static void main(String[] args) {

BankAccount account = new BankAccount("Alice", 1000);

account.deposit(500); // Deposited: $500

account.withdraw(200); // Withdrawn: $200

System.out.println(account.getBalance()); // Output: 1300

// account.balance = 999999; ❌ Error: balance is private

}

}  
  
**💼 More Real-World Examples**

| **Class** | **Encapsulated Fields** | **Accessed via** |
| --- | --- | --- |
| Employee | name, id, salary | getSalary(), setSalary() |
| Student | name, marks, grade | getGrade(), calculateResult() |
| Car | speed, fuelLevel | accelerate(), refuel() |
| LoginManager | username, password | login(), validateCredentials() |

**🧩 Quiz Time (with Answers)**

**1. What is encapsulation in Java?**  
A) Inheriting methods  
B) Hiding internal state using private fields ✅  
C) Using interfaces  
D) Overloading constructors

**2. Which access modifier is most commonly used to achieve encapsulation?**  
A) public  
B) protected  
C) private ✅  
D) static

**3. Can you access a private variable directly outside its class?**  
A) Yes  
B) No ✅

**4. Which of the following is true about encapsulation?**  
A) It makes code less secure  
B) It allows direct access to all variables  
C) It binds data and behavior together ✅  
D) It eliminates the need for constructors

**🎮 Guess the Output Exercise**  
🔹 Exercise 1:  
class Student {

private String name;

public void setName(String name) {

this.name = name;

}

public void printName() {

System.out.println("Name: " + name);

}

}

public class Main {

public static void main(String[] args) {

Student s = new Student();

s.setName("John");

s.printName();

}

}  
  
🔹 Exercise 2:  
class Box {

private int length;

public void setLength(int len) {

if (len > 0)

length = len;

}

public int getLength() {

return length;

}

}

public class Main {

public static void main(String[] args) {

Box b = new Box();

b.setLength(-10);

System.out.println(b.getLength());

}

}

# **Interview kind of questions:**

1. **Can we override a static method in Java?**

**Answer:**  
No, static methods cannot be overridden in Java. Static methods are resolved at compile time, not runtime, and belong to the class, not the instance. If a static method is defined in both the superclass and subclass, it is called method hiding, not overriding.

**Example:**

class Parent {

static void show() {

System.out.println("Parent class show");

}

}

class Child extends Parent {

static void show() {

System.out.println("Child class show");

}

}

public class TestStaticMethod {

public static void main(String[] args) {

Parent.show(); // Output: Parent class show

Child.show(); // Output: Child class show

}

}

Here, the method show() is hidden, not overridden, because it's static.

**2. What is the significance of the super keyword in method overriding?**

**Answer:**  
The super keyword in Java is used to refer to the immediate parent class of the current object. It can be used to call parent class methods and constructors. When overriding a method in the child class, super can be used to invoke the parent class version of the method.

**Example:**

class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

void sound() {

super.sound(); // Calls the method from Animal class

System.out.println("Dog barks");

}

}

public class TestSuperKeyword {

public static void main(String[] args) {

Dog dog = new Dog();

dog.sound();

// Output:

// Animal makes a sound

// Dog barks

}

}

**3. What is the role of interfaces in achieving polymorphism in Java?**

**Answer:**  
Interfaces in Java play a crucial role in achieving polymorphism. By defining a set of abstract methods in an interface, different classes can implement that interface and provide their own specific implementation for the methods. This allows an object to be treated as an instance of the interface, regardless of which class implements it.

**Example:**

interface Animal {

void sound();

}

class Dog implements Animal {

@Override

public void sound() {

System.out.println("Dog barks");

}

}

class Cat implements Animal {

@Override

public void sound() {

System.out.println("Cat meows");

}

}

public class TestInterfacePolymorphism {

public static void main(String[] args) {

Animal animal = new Dog();

animal.sound(); // Output: Dog barks

animal = new Cat();

animal.sound(); // Output: Cat meows

}

}

In this case, animal can refer to either a Dog or Cat, allowing polymorphism through the interface Animal.

**4. What is method overriding and how does it differ from method overloading?**

**Answer:**

* **Method Overloading:** It occurs when a class has more than one method with the same name but different parameters (in type, number, or both). Overloading is resolved at compile time, making it compile-time polymorphism.
* **Method Overriding:** It occurs when a subclass provides its own implementation of a method that is already defined in the superclass. The method to be called is determined at runtime, making it runtime polymorphism.

**Assignments for this week:  
🔐 Assignment 1: Student Marks Manager**

**Problem Statement:**  
Create a class Student with the following fields (as private):

* name
* rollNumber
* marks[] (array of 5 subjects)

Add the following:

* A constructor to initialize name and rollNumber.
* A method setMarks(int[] marks) to store marks (only if all values are between 0–100).
* A method getAverage() to return the average of the marks.
* A method getGrade() that returns:
  + "A" if avg ≥ 90
  + "B" if avg ≥ 75
  + "C" if avg ≥ 50
  + "F" otherwise

Input:  
Name: John

Roll Number: 101

Marks: [90, 85, 78, 88, 92]  
  
Output:  
Student: John (101)

Average: 86.6

Grade: B  
  
🏦 **Assignment 2: Bank Account System  
Problem Statement:**  
Create a class BankAccount with the following:

* private fields: accountNumber, accountHolder, balance
* Constructor to initialize the fields
* Public methods:
  + deposit(double amount)
  + withdraw(double amount)
  + getBalance()

Use validation to ensure:

* Balance can’t go negative
* Deposit/withdraw amount should be > 0

Input/Output Example:  
Deposit: $500

Withdraw: $200

Balance: $300  
  
🖨️ **Assignment 3: Print Utility (Method Overloading)  
Problem Statement:**  
Create a class Printer with an overloaded print() method:

* print(String text)
* print(int number)
* print(double number)
* print(String text, int times) – prints text multiple times

Test each method with appropriate inputs.

Input:  
Printer p = new Printer();

p.print("Hello");

p.print(100);

p.print(99.99);

p.print("Java", 3);  
  
Output:  
Hello

100

99.99

Java

Java

Java

**🚗 Assignment 4: Transport Fare Calculator**

**🔹 Objective:**

Demonstrate method overriding by implementing different fare calculation logic for various transport types.

**📝 Problem Statement:**

Create a base class Transport and derive classes Bus, Train, and Taxi. Each subclass should override the calculateFare(int distance) method with its own pricing logic.

**🔧 Requirements:**

1. **Base Class: Transport**
   * Method: public double calculateFare(int distance)
   * Default logic: fare = distance \* 1.0
2. **Derived Classes:**
   * **Bus**: fare = distance \* 0.5
   * **Train**: fare = distance \* 0.8
   * **Taxi**: fare = base fare 50 + distance \* 2.0
3. Create a main() method to:
   * Instantiate each type
   * Call calculateFare() for a sample distance
   * Print the output with transport type

**📥 Sample Input:**

int distance = 20;

**📤 Expected Output:**

Bus Fare for 20 km: ₹10.0

Train Fare for 20 km: ₹16.0

Taxi Fare for 20 km: ₹90.0