Polymorphism in Java (OOP's Method's)

1. Introduction

Definition

Polymorphism = "Poly" (many) + "Morph" (forms)

> ‼ The ability of an object to take on multiple forms, allowing the same method or object to behave differently based on the context.

Real-World Analogy

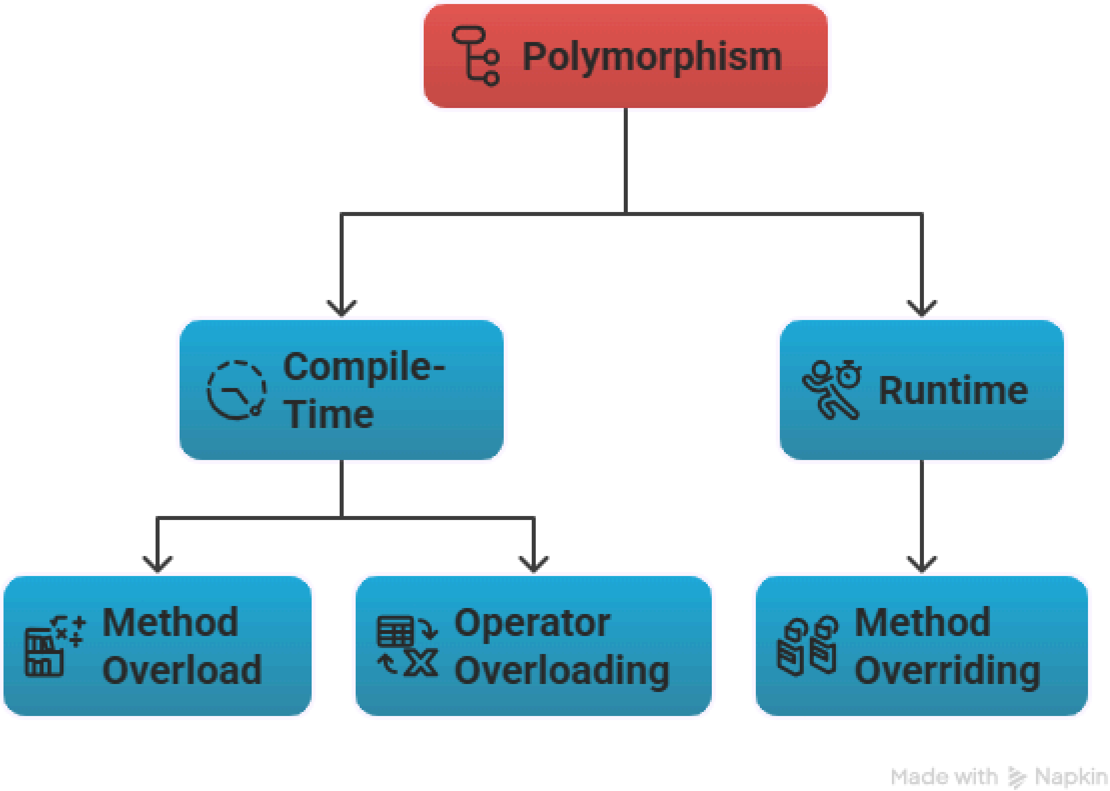
```
REAL-WORLD EXAMPLE
```

```
│  A person can be:                                          │
│  • A student in college                                    │
│  • A son/daughter at home                                  │
│  • An employee at work                                     │
│  • A customer in a shop                                    │
│                                                            │
│  Same person → Different behaviors in different contexts   │
```

Benefits of Polymorphism

```
✅ Code Reusability
✅ Flexibility and Extensibility
✅ Simplified Code Maintenance
✅ Loose Coupling
✅ Supports "Program to Interface" principle
```

2. Types of Polymorphism



1. Compile-time Polymorphism (Static Binding)

> ‼ This occurs when the compiler determines which method to call at compile time. This is achieved through Method Overloading.

[A] Method Overloading

> ‼ Defining multiple methods in the same class with the same name but different parameters.

Rules for Overloading:

1. Change the number of arguments.
2. Change the data type of arguments.
3. Change the order of arguments.
4. *Note:* Changing only the *return type* is not sufficient for overloading.

Code Example: Overloading

```java
class MathOperations {
    // Method 1: Two integer parameters
    public int add(int a, int b) {
        return a + b;
    }

    // Method 2: Three integer parameters
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method 3: Two double parameters
    public double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        MathOperations math = new MathOperations();

        System.out.println(math.add(5, 10));       // Calls Method 1
        System.out.println(math.add(5, 10, 15));   // Calls Method 2
        System.out.println(math.add(5.5, 2.3));    // Calls Method 3
    }
}
```

[B] Operator Overloading

```
OPERATOR OVERLOADING IN JAVA
```

```
| Java does NOT support user-defined operator overloading       |
|                                                               |
| However, Java has ONE built-in overloaded operator:           |
|                                                               |
|  + (Plus Operator)                                            |
|  • Addition for numbers: 5 + 3 = 8                            |
|  • Concatenation for Strings: "Hello" + "World" = "HelloWorld"|
```

Example -

```java
public class OperatorOverloadingExample {
    public static void main(String[] args) {
        // + operator for arithmetic addition
        int sum = 10 + 20;
        System.out.println("Sum: " + sum);  // Output: 30

        // + operator for String concatenation
        String firstName = "John";
        String lastName = "Doe";
        String fullName = firstName + " " + lastName;
        System.out.println("Full Name: " + fullName);  // Output: John Doe

        // Mixed usage
        String result = "Sum is: " + (10 + 20);
        System.out.println(result);  // Output: Sum is: 30
    }
}
```

2. Runtime Polymorphism (Dynamic Polymorphism)

Method Overriding

Definition

> !! A subclass provides a **specific implementation** of a method that is **already defined** in its parent class.

Rules for Method Overriding

```
| ✅ Same method name                                             |
| ✅ Same parameters (number, type, order)                        |
| ✅ Same return type (or covariant return type)                  |
| ✅ IS-A relationship required (inheritance)                     |
| ✅ Access modifier can be same or less restrictive              |
| ❌ Cannot override static methods (hiding, not overriding)      |
| ❌ Cannot override final methods                                |
| ❌ Cannot override private methods                              |
| ❌ Cannot have more restrictive access modifier                 |
```

Example 1: Basic Method Overriding

```java
// Parent class
class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }

    public void eat() {
        System.out.println("Animal is eating");
    }
}

// Child class 1
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks: Woof! Woof!");
    }
    @Override
    public void eat() {
        System.out.println("Dog is eating bones");
    }

    public void fetch() {
        System.out.println("Dog is fetching the ball");
    }
}

// Child class 2
class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat meows: Meow! Meow!");
    }

    @Override
    public void eat() {
        System.out.println("Cat is eating fish");
    }

    public void scratch() {
        System.out.println("Cat is scratching");
    }
}

// Main class
public class MethodOverridingDemo {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Animal dog = new Dog();      // Upcasting
        Animal cat = new Cat();      // Upcasting

        System.out.println("=== Animal ===");
        animal.makeSound();
        animal.eat();

        System.out.println("\n=== Dog (as Animal reference) ===");
        dog.makeSound();    // Dog's overridden method
        dog.eat();          // Dog's overridden method
        // dog.fetch();     // ERROR: Animal reference can't access Dog-specific

        System.out.println("\n=== Cat (as Animal reference) ===");
        cat.makeSound();    // Cat's overridden method
        cat.eat();          // Cat's overridden method
    }
}
```

Output -

```
=== Animal ===
Animal makes a sound
Animal is eating

=== Dog (as Animal reference) ===
Dog barks: Woof! Woof!
Dog is eating bones

=== Cat (as Animal reference) ===
Cat meows: Meow! Meow!
Cat is eating fish
```

Comparison: Overloading vs. Overriding

Feature	Method Overloading	Method Overriding
Type	Compile-time Polymorphism	Runtime Polymorphism
Scope	Within the same class	Across Parent-Child classes (Inheritance)
Method Signature	Name same, parameters different	Name same, parameters same
Return Type	Can be different	Must be same (or covariant)
Binding	Static Binding	Dynamic Binding
Private/Static	Can be overloaded	Cannot be overridden

Advantages of Polymorphism

1. **Code Reusability:** You can write generic code that works with a Parent class type, and it will automatically handle any new Child classes added in the future.
2. **Flexibility:** It supports the **Open/Closed Principle** (Open for extension, closed for modification). You can add new animal types (e.g., `Lion`) without changing the logic that makes animals speak.
3. **Cleaner Code:** Reduces `if-else` or `switch` statements to check for types