16.3 Polymorphism

📘 Polymorphism in Java

✨ 1. Introduction to Polymorphism

What is Polymorphism?

- **Polymorphism** comes from Greek: *"poly"* = many, *"morph"* = forms.
- It means **one interface, many implementations**.
- In Java, polymorphism allows **objects to be treated as instances of their parent class**, even if they are actually instances of a subclass.

Why is Polymorphism Important?

- Promotes **code reusability** and **flexibility**.
- Helps in **method overriding** and **runtime method binding**.
- Supports **dynamic method dispatch**, a core OOP concept.

🧱 2. Types of Polymorphism in Java

A. Compile-Time Polymorphism (Static Binding)

- Also known as **Method Overloading**.
- Method resolution is done at **compile time**.
- **Same method name**, but different parameters (type/number/order).

Example:

```java
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;
    }
}
```

Key Points:

- Based on **method signature**.
- Happens during **compilation**.
- Faster than runtime polymorphism.

B. Runtime Polymorphism (Dynamic Binding)

- Achieved using **Method Overriding**.
- Method resolution is done at **runtime**.
- Requires **inheritance** and **upcasting**.

Example:

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

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

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

public class TestPolymorphism {
    public static void main(String[] args) {
        Animal a;

        a = new Dog();
        a.sound();  // Dog barks

        a = new Cat();
        a.sound();  // Cat meows
    }
}
```

Key Points:

- Uses **inheritance** and **method overriding**.
- Object is accessed via a **parent class reference**.
- Resolved during **runtime**.
- Supports **dynamic method dispatch**.

🧠 3. Method Overloading vs Method Overriding

Feature	Overloading	Overriding
Class	Same class	Subclass
Parameters	Must be different	Must be same
Return Type	Can be different	Must be same (or covariant)
Access Modifier	Any	Cannot be more restrictive
Static/Final Method	Can be overloaded	Cannot be overridden
Binding Time	Compile-time	Runtime

🧰 4. Practical Use Cases of Polymorphism

- Designing **flexible APIs**.
- Implementing **plugin systems** or **frameworks**.
- Using **interfaces and abstract classes** to define a common contract.
- In **collections**: `List<String>`, `ArrayList<Integer>`, etc., all use polymorphism.

⚠️ 5. Things to Remember

- **Only non-static methods** are polymorphic.
- Java always calls **overridden methods** based on the **actual object**, not the reference type.
- Constructors **cannot be overridden** but can be **overloaded**.

📝 6. Summary

- **Polymorphism** is a key concept in OOP that enables flexibility and reuse.
- Java supports **compile-time** and **runtime** polymorphism.
- Through **overloading** and **overriding**, Java lets developers write more maintainable, scalable code.
