**What is Polymorphism?**

**Polymorphism** is one of the core concepts in Object-Oriented Programming (OOP). The word "polymorphism" comes from the Greek words ***poly*** (meaning "many") and ***morph*** (meaning "form"), so it literally means "many forms." In programming, polymorphism refers to the ability of one entity (such as a method or an object) to take on **many forms**.

In Java, polymorphism mainly comes in two forms:

1. **Compile-time Polymorphism** (Method Overloading)
2. **Runtime Polymorphism** (Method Overriding)

Let's go through both types with examples to understand where exactly polymorphism happens in the code.

**1. Compile-time Polymorphism (Method Overloading)**

**Method overloading** happens when there are multiple methods with the **same name** but with **different parameters** (either in the number of parameters or type of parameters). The method that gets called is determined at **compile time** based on the method signature.

**Example of Compile-time Polymorphism:**

class Printer {

// Method to print a string

void print(String message) {

System.out.println("Printing string: " + message);

}

// Method to print an integer

void print(int number) {

System.out.println("Printing integer: " + number);

}

// Method to print a double

void print(double value) {

System.out.println("Printing double: " + value);

}

}

public class Main {

public static void main(String[] args) {

Printer printer = new Printer();

printer.print("Hello, world!"); // Calls print(String)

printer.print(123); // Calls print(int)

printer.print(45.67); // Calls print(double)

}

}

**Where Polymorphism Happens:**

* In the code above, we have **three methods** in the Printer class named print but with **different parameter types** (String, int, and double).
* When you call printer.print(), Java knows which method to call based on the **type and number of arguments** passed to it.
* This is **compile-time polymorphism** because the method to be called is decided at the **compile time**, based on the method signature.

**2. Runtime Polymorphism (Method Overriding)**

**Method overriding** occurs when a subclass provides its own implementation of a method that is already defined in its superclass. The method in the subclass **overrides** the method in the superclass, and the appropriate method is called based on the **actual object type** (the object created at runtime), not the reference type.

**Example of Runtime Polymorphism:**

class Animal {

// Method in the parent class

void makeSound() {

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

}

}

class Dog extends Animal {

// Method in the child class overriding the parent class method

@Override

void makeSound() {

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

}

}

class Cat extends Animal {

// Method in the child class overriding the parent class method

@Override

void makeSound() {

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

}

}

public class Main {

public static void main(String[] args) {

Animal myAnimal = new Animal(); // Animal object

Animal myDog = new Dog(); // Dog object

Animal myCat = new Cat(); // Cat object

myAnimal.makeSound(); // Output: Animal makes a sound

myDog.makeSound(); // Output: Dog barks

myCat.makeSound(); // Output: Cat meows

}

}

**Where Polymorphism Happens:**

* In the above example, both the Dog and Cat classes **override** the makeSound() method from the Animal class.
* When we use the Animal reference type to point to objects of Dog and Cat, we achieve **runtime polymorphism**.
* Even though the reference variable is of type Animal, the method that is executed is determined by the **actual object type** (either Dog or Cat) at **runtime**.
  + If myDog refers to a Dog object, myDog.makeSound() calls the overridden method in Dog, not Animal.
  + If myCat refers to a Cat object, myCat.makeSound() calls the overridden method in Cat, not Animal.

**Key Points About Polymorphism:**

1. **Compile-time Polymorphism (Method Overloading)**:
   * Occurs when multiple methods with the same name but different signatures (parameter types or number of parameters) exist in a class.
   * The correct method signature is selected at **compile-time (and not the implementation i.e. method belongs to which class)** based on the arguments passed.
   * It is also known as **method overloading**.
2. **Runtime Polymorphism (Method Overriding)**:
   * Occurs when a subclass provides its own implementation of a method that is already defined in its superclass.
   * The method is selected at **runtime** based on the **actual object type** (not the reference type).
   * It is also known as **method overriding** and is typically used with inheritance.

**Benefits of Polymorphism:**

* **Flexibility**: Polymorphism allows you to write more flexible and reusable code. For instance, if you have a method that takes a Shape object, you can pass in any subclass of Shape (e.g., Circle, Rectangle) and the correct method will be called.
* **Maintainability**: It helps in maintaining code by allowing you to extend functionality without changing existing code. You can add new subclasses that override methods and introduce new behavior.

**Real-life Analogy:**

Imagine you have a **remote control** for several different types of **electronic devices**—a **TV**, a **fan**, and an **AC (Air Conditioner)**. Each device has a **power button**. The button for **turning on** is the same (i.e., **polymorphic method**), but each device performs a different action when the button is pressed:

* **TV** starts showing channels.
* **Fan** starts rotating the blades.
* **AC** starts cooling the room.

Even though each device has a **power button**, the action that occurs is different for each type of device. This is similar to **runtime polymorphism**, where a **method** (powerOn()) behaves differently depending on the actual object type (TV, Fan, AC).

**Summary of Polymorphism:**

* **Polymorphism** allows you to use a **single interface** to represent different underlying forms (objects).
* **Compile-time polymorphism** (method overloading) is resolved during **compilation** based on method signature.
* **Runtime polymorphism** (method overriding) is resolved during **runtime** based on the actual object type.

Polymorphism enables more **flexible**, **maintainable**, and **extensible** code.