**Polymorphism**

is a fundamental concept in object-oriented programming (OOP) that allows objects of different classes to be

treated as objects of a common interface or base class. The term "polymorphism" comes from the Greek words "poly," meaning many, and "morph," meaning form. In the context of programming, polymorphism provides a way to write code that works with objects of various types in a unified manner.

**There are two main types of polymorphism in Java:**

**Compile-Time Polymorphism (Method Overloading):**

Compile-time polymorphism, also known as method overloading, occurs when multiple methods in the same class have the same name but different parameters (either a different number of parameters or different types).

The appropriate method is chosen by the compiler based on the method signature during compilation.

Example:

**public class Calculator {**

**// Method Overloading**

**public int add(int a, int b) {**

**return a + b;**

**}**

**public double add(double a, double b) {**

**return a + b;**

**}**

**}**

**Run-Time Polymorphism (Method Overriding):**

Run-time polymorphism, also known as method overriding, occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

The appropriate method is chosen at runtime based on the actual type of the object.

Example:

**// Superclass**

**class Animal {**

**void makeSound() {**

**System.out.println("Some generic sound");**

**}**

**}**

**// Subclass**

**class Dog extends Animal {**

**// Method Overriding**

**void makeSound() {**

**System.out.println("Bark!");**

**}**

**}**

In this example, if you have an object of type Animal but it refers to an instance of Dog, calling makeSound() will execute the overridden method in the Dog class.

Polymorphism provides several benefits, including:

**Code Reusability:** Polymorphism allows code to be written in a generic way, making it more reusable and adaptable to different types of objects.

**Flexibility:** Polymorphism enables the creation of code that can work with objects of various types, providing flexibility in design.

**Abstraction:** Polymorphism allows you to program at a higher level of abstraction, focusing on the behavior of objects rather than their specific types.

In summary, polymorphism in Java enables the ability to use a single interface to represent different types of objects, either through method overloading or method overriding. This concept plays a crucial role in achieving flexibility, reusability, and abstraction in object-oriented programming.

**A real-life** example of polymorphism can be found in the concept of a "Shape" in a drawing or graphic application. In such an application, various shapes like circles, rectangles, and triangles can be drawn on the screen. Each shape has a common behavior, such as drawing itself, even though the specific implementation details differ for each type of shape. This scenario is a classic example of polymorphism through method overriding.

**Let's create a simplified example in Java to illustrate this:**

// Common interface for shapes

interface Shape {

void draw();

}

// Circle class implementing the Shape interface

class Circle implements Shape {

private double radius;

public Circle(double radius) {

this.radius = radius;

}

@Override

public void draw() {

System.out.println("Drawing a circle with radius " + radius);

// Additional circle-specific drawing logic...

}

}

// Rectangle class implementing the Shape interface

class Rectangle implements Shape {

private double length;

private double width;

public Rectangle(double length, double width) {

this.length = length;

this.width = width;

}

@Override

public void draw() {

System.out.println("Drawing a rectangle with length " + length + " and width " + width);

// Additional rectangle-specific drawing logic...

}

}

In this example:

The Shape interface defines a common method draw() for all shapes.

The Circle and Rectangle classes implement the Shape interface and provide specific implementations for the draw() method.

Now, let's use these classes in a drawing application:

public class DrawingApp {

public static void main(String[] args) {

// Polymorphism in action

Shape circle = new Circle(5.0);

Shape rectangle = new Rectangle(4.0, 6.0);

// Draw different shapes without knowing their specific types

drawShape(circle);

drawShape(rectangle);

}

// Method that can draw any shape implementing the Shape interface

private static void drawShape(Shape shape) {

shape.draw();

}

}

In this example, the drawShape method accepts any object that implements the Shape interface. The method doesn't need to know the specific type of shape; it simply calls the draw() method, and the appropriate implementation is chosen at runtime based on the actual type of the object.

This demonstrates polymorphism in action, where different types of shapes can be treated uniformly through a common interface, allowing for code reuse and flexibility in handling various shapes in a drawing application.