Polymorphism

Polymorphism in Java is a fundamental concept in object-oriented programming (OOP) that allows one interface to be used for a general class of actions. It's a mechanism that allows objects to be treated as instances of their parent class, enabling flexibility and reusability in code.

Key Benefits of Polymorphism

- **Code Reusability**: Common methods can be reused across multiple classes.
- **Flexibility**: Code written to work with superclass types or interfaces can be reused with any subclass or implementing class.
- **Extensibility**: New subclasses or implementations can be added with minimal changes to existing code.

There are two main types of polymorphism in Java:

1. **Compile-time (Static) Polymorphism**: Achieved through method overloading.
2. **Runtime (Dynamic) Polymorphism**: Achieved through method overriding.

1. Compile-Time (Static) Polymorphism

Compile-time polymorphism is achieved by **method overloading**. It is called compile-time polymorphism because the compiler determines which method to execute based on the method signature.

Method Overloading Example

Method overloading allows multiple methods with the same name but different parameter lists in the same class. This lets you perform similar operations using different types or numbers of parameters.

```
class Calculator {
    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
```

```
    }

    // Overloaded method to add two doubles
    public double add(double a, double b) {
        return a + b;
    }
}

public class StaticPolymorphismExample {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();

        System.out.println("Sum of two integers: " + calculator.add(5,
10));
        System.out.println("Sum of three integers: " +
calculator.add(5, 10, 15));
        System.out.println("Sum of two doubles: " +
calculator.add(5.5, 10.5));
    }
}
```

Explanation:

- The `Calculator` class has three `add` methods with different parameter lists.
- The compiler chooses the correct `add` method based on the parameters passed.

2. Runtime (Dynamic) Polymorphism

Runtime polymorphism is achieved through **method overriding**. Here, the JVM decides which method to execute based on the runtime object. This is possible with inheritance, where a subclass provides a specific implementation of a method that already exists in its superclass.

Method Overriding Example

Method overriding allows a subclass to provide a specific implementation of a method already defined in its superclass.

```
class Animal {
    // Method to be overridden
```

```
    public void sound() {
        System.out.println("Some generic animal sound");
    }
}

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

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

public class DynamicPolymorphismExample {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        myDog.sound(); // Outputs: Bark
        myCat.sound(); // Outputs: Meow
    }
}
```

Explanation:

- `Animal` is the superclass with a generic `sound()` method.
- `Dog` and `Cat` are subclasses that override `sound()` with specific implementations.
- At runtime, the JVM calls the `sound()` method of the actual object type (`Dog` or `Cat`), demonstrating runtime polymorphism.

3. Polymorphism with Interfaces

Polymorphism can also be achieved using interfaces. When a class implements an interface, it can define different implementations for the methods declared in the interface.

Interface Polymorphism Example

```
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing a circle");
    }
}

class Rectangle implements Shape {
    public void draw() {
        System.out.println("Drawing a rectangle");
    }
}

public class InterfacePolymorphismExample {
    public static void main(String[] args) {
        Shape shape1 = new Circle();
        Shape shape2 = new Rectangle();

        shape1.draw(); // Outputs: Drawing a circle
        shape2.draw(); // Outputs: Drawing a rectangle
    }
}
```

Explanation:

- The `Shape` interface declares a `draw()` method.
- `Circle` and `Rectangle` classes implement the `Shape` interface and provide their specific implementations of `draw()`.
- The `draw()` method is called polymorphically on each `Shape` reference.

4. Polymorphism with Abstract Classes

Abstract classes also support polymorphism, where subclasses provide their implementations of the abstract methods defined in the abstract superclass.

Abstract Class Polymorphism Example

```
abstract class Vehicle {
    abstract void start();
}

class Car extends Vehicle {
    @Override
    void start() {
        System.out.println("Car starts with a key");
    }
}

class Bike extends Vehicle {
    @Override
    void start() {
        System.out.println("Bike starts with a button");
    }
}

public class AbstractPolymorphismExample {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        Vehicle myBike = new Bike();

        myCar.start(); // Outputs: Car starts with a key
        myBike.start(); // Outputs: Bike starts with a button
    }
}
```

Explanation:

- `Vehicle` is an abstract class with an abstract `start()` method.
- `Car` and `Bike` subclasses override the `start()` method with their specific implementations.
- Polymorphism allows us to call the `start()` method on a `Vehicle` reference, but the actual method that gets called depends on the runtime object (`Car` or `Bike`).

5. Casting and `instanceof` with Polymorphism

In polymorphism, you may sometimes need to check the actual type of an object to perform specific operations. The `instaceof` operator and casting allow you to handle objects of different types.

Example with Casting and `instanceof`

```
class Animal {
    public void makeSound() {
        System.out.println("Some animal sound");
    }
}

class Dog extends Animal {
    public void fetch() {
        System.out.println("Dog is fetching");
    }
}

public class InstanceofExample {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();

        // Check if myAnimal is an instance of Dog
        if (myAnimal instanceof Dog) {
            Dog myDog = (Dog) myAnimal; // Downcast to Dog
            myDog.fetch(); // Outputs: Dog is fetching
        }
    }
}
```

Explanation:

- `instanceof` checks if `myAnimal` is a `Dog` before downcasting.
- Casting allows access to the `fetch()` method, which is specific to the `Dog` class.

Java Interfaces and Abstract Classes

In Java, **interfaces** and **abstract classes** are mechanisms for achieving abstraction, which allows developers to define "what to do" (methods) without specifying "how to do it" (implementations). While both have overlapping purposes, they differ in their use cases, structure, and capabilities.

---

1. Abstract Class

An **abstract class** is a blueprint for other classes. It can have both abstract methods (without a body) and concrete methods (with a body). You cannot instantiate an abstract class directly; it must be extended by a subclass.

Key Characteristics:

1. **Abstract Methods**: These are methods without a body, meant to be implemented by subclasses.
   Example: `abstract void display();`
2. **Concrete Methods**: An abstract class can have methods with implementations that can be inherited by subclasses.
3. **Constructors**: Abstract classes can have constructors, which can be called during object creation of concrete subclasses.
4. **Instance Variables**: Abstract classes can have fields and can define their visibility (`private`, `protected`, etc.).
5. **Access Modifiers**: Abstract classes can have methods with any access modifier (public, protected, private).
6. **Inheritance**: A class can inherit only one abstract class (single inheritance).

When to Use Abstract Classes:

- When you want to provide a common base class for related classes.
- When you want to share code (method implementations) among subclasses.
- When the "is-a" relationship holds strongly, e.g., a `Car` is-a `Vehicle`.

Example:

```
abstract class Animal {
    String name;

    // Constructor
    Animal(String name) {
        this.name = name;
    }

    // Abstract method
    abstract void makeSound();

    // Concrete method
    void sleep() {
        System.out.println(name + " is sleeping.");
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name);
    }

    @Override
    void makeSound() {
        System.out.println(name + " says Woof!");
    }
}
```

2. Interface

An **interface** is a contract or a specification that defines a set of methods that a class must implement. It cannot have any implementation (prior to Java 8). From Java 8 onwards, it can also include default and static methods with implementations.

Key Characteristics:

1. **Abstract Methods (Pre-Java 8)**: All methods are implicitly `public` and `abstract` by default.
2. **Default Methods (Java 8+)**: Interfaces can define default methods with a body. These provide a way to evolve interfaces without breaking existing implementations. Example: `default void log(String message) { System.out.println(message); }`
3. **Static Methods (Java 8+)**: Interfaces can have static methods with a body. These are not inherited by implementing classes.
4. **Private Methods (Java 9+)**: Interfaces can define private helper methods that default or static methods can use.
5. **No State**: Interfaces cannot have instance variables but can have `public static final` constants.
6. **Multiple Inheritance**: A class can implement multiple interfaces, achieving multiple inheritance in Java.

When to Use Interfaces:

- When you want to define a contract for classes with unrelated hierarchies.
- When you want multiple inheritance for behaviors.
- When you want to achieve loose coupling between components.

Example:

```
interface Flyable {
    void fly();

    default void takeOff() {
        System.out.println("Taking off...");
    }

    static void emergencyLanding() {
        System.out.println("Performing an emergency landing!");
    }
}

class Airplane implements Flyable {
    @Override
    public void fly() {
        System.out.println("The airplane is flying.");
    }
```

```
}
```

Comparison Between Abstract Classes and Interfaces

Feature	Abstract Class	Interface
Methods	Can have both abstract and concrete methods.	Can have abstract methods (default from Java 8 onwards).
Default Implementation	Yes, for concrete methods.	Yes, for default and static methods (Java 8+).
Constructors	Can have constructors.	Cannot have constructors.
Fields	Can have instance variables.	Only `public static final` constants.
Access Modifiers	Methods can be `public`, `protected`, `private`.	Methods are `public` by default.
Multiple Inheritance	Not supported.	Supported via multiple interfaces.
Purpose	Used for sharing code and creating a base class.	Used for defining contracts.
Relationship	Represents an "is-a" relationship.	Represents a "can-do" or "behavioral" relationship.

Best Practices

- **Use Abstract Classes:** When you need shared code or a common base implementation.
- **Use Interfaces:** When designing loosely coupled, scalable systems with unrelated classes.
- **Combine Both:** An abstract class can implement an interface to combine the benefits of both.

CodInClub
Powered By BridgeLabz

Interface vs Class in Java

Interfaces and classes are fundamental building blocks in Java. While both are used to define types and structure programs, they serve different purposes and have distinct characteristics.

Key Differences Between Interface and Class

Feature	Interface	Class
Purpose	Defines a contract or a set of rules for implementing classes.	Defines the blueprint for objects, encapsulating data and behavior.
Inheritance	Allows multiple inheritance; a class can implement multiple interfaces.	Supports single inheritance; a class can extend only one other class.
Abstract Methods	Contains abstract methods by default (prior to Java 8). From Java 8+, supports default and static methods with implementation.	Can contain both abstract and concrete methods (if it is an abstract class). Regular classes can have only concrete methods.
Access Modifiers for Methods	Methods are `public` and `abstract` by default (can also include `default`, `static`, and `private` methods in newer versions).	Methods can have any access modifier: `public`, `protected`, `private`, or package-private.
Instance Variables	Cannot have instance variables; only `public static final` constants are allowed.	Can have instance variables with any access modifier (`private`, `protected`, etc.).

Constructors	Cannot have constructors.	Can have constructors to initialize objects.
Multiple Inheritance	Achieves multiple inheritance through implementation of multiple interfaces.	Does not support multiple inheritance (directly).
Static Methods	Allowed (from Java 8).	Allowed.
Fields	Only `public static final` constants are allowed.	Can have mutable fields of any type and access level.
Relationship	Represents a "can-do" or "behavioral" relationship.	Represents an "is-a" relationship in inheritance.
Instantiation	Cannot be instantiated directly.	Can be instantiated unless it is an abstract class.

Use Cases for Interfaces

1. **Defining Contracts**: Use interfaces to specify a contract that unrelated classes can follow, e.g., `Comparable`, `Runnable`.
2. **Achieving Multiple Inheritance**: Interfaces allow a class to implement multiple behaviors.
3. **Loose Coupling**: Interfaces provide abstraction, ensuring loose coupling between components.

Use Cases for Classes

1. **Encapsulation**: Use classes to encapsulate data and related methods.
2. **Inheritance**: Create a base class to share code across related classes.
3. **Object Creation**: Use classes as blueprints to create objects with properties and methods.

Example: Interface vs Class

Interface Example

```
interface Flyable {
    void fly(); // Abstract method
    default void takeOff() {
        System.out.println("Taking off...");
    }
}

class Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("The bird is flying.");
    }
}
```

Class Example

```
class Animal {
    String name;
    Animal(String name) {
        this.name = name;
    }
    void eat() {
        System.out.println(name + " is eating.");
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name);
    }
    void bark() {
        System.out.println(name + " says Woof!");
    }
}
```

When to Use Interface vs Class

- **Use an Interface**:
  - When multiple unrelated classes need to implement a common set of methods.
  - When you want to enforce a specific behavior (e.g., `Serializable` or `Runnable`).
  - When designing systems that require loose coupling.
- **Use a Class**:
  - When you need to define concrete objects with state and behavior.
  - When code reuse is required through inheritance.
  - When encapsulating data and logic into a single unit.