### Inheritance in Java: Concept + 5 Types

Inheritance is an **OOP (Object-Oriented Programming)** concept where a **child class (subclass)** inherits properties and behaviors from a **parent class (superclass)**.

#### Key Terms

- **Superclass (Parent)** – The class being inherited from.
- **Subclass (Child)** – The class that inherits.
- **Reusability** – Avoids rewriting code (DRY Principle).

---

## 5 Types of Inheritance in Java

### 1. Single Inheritance

- A **child class inherits from one parent class**.
- **Syntax**:

```
class Parent { }
```

- `class Child extends Parent { }`
- **Example**:

```
class Animal { void eat() { System.out.println("Eating..."); } }
```

- `class Dog extends Animal { void bark() { System.out.println("Barking!"); } }`

---

### 2. Multilevel Inheritance

- A **child class inherits from another child class** (chain-like structure).
- **Syntax**:

```
class A { }
class B extends A { }
```

- `class C extends B { }`
- **Example**:

```
class Grandfather { void old() { System.out.println("Old age"); } }
class Father extends Grandfather { void middle() { System.out.println("Middle age"); } }
```

- `class Son extends Father { void young() { System.out.println("Young age"); } }`

---

### 3. Hierarchical Inheritance

- **Multiple child classes inherit from a single parent.**
- **Syntax**:

```
class Parent { }
class Child1 extends Parent { }
```

- `class Child2 extends Parent { }`
- **Example**:

```
class Vehicle { void run() { System.out.println("Moving"); } }
class Car extends Vehicle { }
```

- `class Bike extends Vehicle { }`

---

### 4. Multiple Inheritance (Not Supported in Java via Classes)

- A **child class inherits from multiple parents** (Java doesn't allow this to avoid the **"Diamond Problem"**).
- **Alternative**: Use **interfaces** (`implements` keyword).
- **Syntax**:

```
interface A { }
interface B { }
```

- `class C implements A, B { }`
- **Example**:

```
interface Father { void beard(); }
interface Mother { void softVoice(); }
class Child implements Father, Mother {
    public void beard() { System.out.println("Has beard"); }
    public void softVoice() { System.out.println("Soft voice"); }
```

- `}`

---

### 5. Hybrid Inheritance (Combination of Multiple Types)

- A **mix of two or more inheritance types** (e.g., Single + Hierarchical).

- **Example**:

```
class A { }
class B extends A { }
class C extends A { }
```

- `class D extends B { }  // Hybrid (Single + Hierarchical)`

---

## Why Java Doesn’t Support Multiple Inheritance with Classes?

❌ **Diamond Problem**:

- If **Class C** extends **Class A** and **Class B**, and both A & B have the **same method**, Java won’t know which one to inherit.
- **Solution**: Use **interfaces** (since they provide **method declarations**, not implementations).

---

## Interview Questions

**Q1: What is the "Diamond Problem"?**

- When **two parent classes** have the **same method**, causing ambiguity in the child class.

**Q2: How to achieve multiple inheritance in Java?**

- Using **interfaces** (`implements` keyword).

**Q3: Which inheritance type is most used in Java?**

- **Single & Multilevel** (common in real-world apps).

---

### Summary

| Type | Supported in Java? | Example |
|---|---|---|

| | | |
|---|---|---|
| **Single** | ✅ Yes | `Dog extends Animal` |
| **Multilevel** | ✅ Yes | `Son → Father → Grandfather` |
| **Hierarchical** | ✅ Yes | `Car, Bike → Vehicle` |
| **Multiple (via Interfaces)** | ✅ Yes (only with interfaces) | `class C implements A, B` |
| **Hybrid** | ✅ Yes (combination) | `Single + Hierarchical` |

**Press Enter** for the next topic! 🚀

### Encapsulation in Java

**Encapsulation** is one of the four fundamental OOP (Object-Oriented Programming) principles. It is the concept of **bundling the data (variables) and the methods (functions) that operate on the data into a single unit**, or **class**, and restricting access to some of the object's components. This is usually done by making some fields or methods private and providing public getter and setter methods to access or modify these private fields.

#### Key Points of Encapsulation

1. **Data Hiding**: By making class fields private, you restrict direct access to them. This helps protect the internal state of an object.

2. **Control Over Data**: By using getter and setter methods, you can control how the data is accessed or modified. For instance, you can add validation logic in the setter methods.

3. **Maintainability**: Encapsulation improves code maintainability and readability by providing clear access to an object's properties and ensuring that they are accessed

or modified in a controlled manner.

4. **Loose Coupling**: Encapsulation allows for a more modular approach where changes to internal object implementation do not affect external components using the object.

---

### How to Implement Encapsulation in Java

In Java, encapsulation is implemented using **private** fields and **public getter and setter** methods. Here's a basic example:

```
public class Person {



    // Private fields (data hiding)

    private String name;

    private int age;



    // Public getter for 'name'

    public String getName() {

        return name;

    }

    // Public setter for 'name'

    public void setName(String name) {

        this.name = name;
```

```
    }

    // Public getter for 'age'

    public int getAge() {

        return age;

    }

    // Public setter for 'age' with validation

    public void setAge(int age) {

        if (age > 0) { // Validation logic

            this.age = age;

        } else {

            System.out.println("Age must be positive!");

        }

    }

}
```

**Usage of Encapsulation**

```
public class TestEncapsulation {

    public static void main(String[] args) {

        // Create a new Person object

        Person person = new Person();
```

```java
        // Use setter methods to set data
        person.setName("John");
        person.setAge(30);

        // Use getter methods to retrieve data
        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());
    }
}
```

In the above example:

- The `name` and `age` fields are private, meaning they can't be accessed directly outside the `Person` class.

- The getter and setter methods allow controlled access to the fields. The setter for `age` includes basic validation to ensure that the age is positive.

---

**Real-Time Example of Encapsulation in Spring Boot**

Let's take a real-time example of **encapsulation** in a Spring Boot application. Suppose we have a **User** entity, where we need to encapsulate the user's information and provide validation when setting the data.

**Step 1: Define the User Entity**

```
import javax.persistence.Entity;

import javax.persistence.Id;

@Entity

public class User {

    @Id

    private Long id;

    private String username;

    private String password;



    // Constructor

    public User() {}

    // Getters and setters (encapsulation)

    public Long getId() {

        return id;

    }
```

```
    public void setId(Long id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        if (username.length() > 3) {  // Validation example
            this.username = username;
        } else {
            throw new IllegalArgumentException("Username must be at
least 4 characters long.");
        }
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        if (password.length() > 5) {  // Validation example
```

```
            this.password = password;
        } else {
            throw new IllegalArgumentException("Password must be at least 6 characters long.");
        }
    }
}
```

In this example, the **User** class has private fields (id, username, and password), and public getter and setter methods. The setters for username and password include basic validation to ensure the data is correct.

**Step 2: User Service Layer**

Now, in the **service layer**, we can use this User entity and perform business logic while maintaining encapsulation.

```
import org.springframework.stereotype.Service;
@Service
public class UserService {

    public User createUser(String username, String password) {
        User user = new User();
        // Setting values using setter methods (with encapsulation)
        user.setUsername(username);
```

```
        user.setPassword(password);

        // Here you could save the user to a database or perform
further logic
        return user;
    }

    public String getUserInfo(User user) {
        // Using getter methods to retrieve data
        return "Username: " + user.getUsername() + ", Password: " +
user.getPassword();
    }
}
```

In this `UserService`, we create a `User` object and set the `username` and `password` using the setter methods. The service layer doesn't need to know the details about how the validation works inside the `User` class, which is an example of **data hiding**. We also retrieve the `username` and `password` through getter methods.

**Step 3: Controller Layer**

Finally, let's define a controller to handle HTTP requests:

```
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.web.bind.annotation.*;

@RestController

@RequestMapping("/users")

public class UserController {



    @Autowired

    private UserService userService;

    @PostMapping("/create")

    public String createUser(@RequestParam String username,
@RequestParam String password) {

        try {

            User user = userService.createUser(username, password);

            return "User created successfully: " +
user.getUsername();

        } catch (IllegalArgumentException e) {

            return "Error: " + e.getMessage();

        }

    }

}
```

In the `UserController`, we expose an endpoint to create a new user. The controller calls the `createUser` method from the `UserService` and handles the result. If the validation fails (e.g., if the username is too short), an exception is thrown, and the error message is returned.

### Polymorphism in Java

**Polymorphism is one of the four fundamental principles of Object-Oriented Programming (OOP). It allows an object to take on multiple forms. In Java, polymorphism allows you to use a single interface or method to represent different types of objects or actions.**

### Types of Polymorphism in Java

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

---

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

**Method Overloading occurs when a class has more than one method with the same name, but different parameters (either in number or type). The appropriate method is chosen at compile-time based on the method signature (the number and type of parameters).**

**Key Characteristics:**

- **Achieved by having multiple methods with the same name but different parameter lists.**
- **Decided at compile time (i.e., static polymorphism).**
- **It helps to increase the readability of the program.**

**Example:**
**java**
**CopyEdit**

```java
class Calculator {
```

```
    // Overloaded method for adding two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method for adding three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method for adding two floating-point numbers
    public double add(double a, double b) {
        return a + b;
    }
}

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

        System.out.println("Sum of two integers: " +
calculator.add(10, 20));      // Calls first method
        System.out.println("Sum of three integers: " +
calculator.add(10, 20, 30)); // Calls second method
        System.out.println("Sum of two doubles: " +
calculator.add(10.5, 20.5));    // Calls third method
    }
}
```

**Output:**

yaml
CopyEdit
```
Sum of two integers: 30
Sum of three integers: 60
Sum of two doubles: 31.0
```

**In this example, the `add` method is overloaded with different parameters, demonstrating compile-time polymorphism.**

---

## 2. Runtime Polymorphism (Method Overriding)

**Method Overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. This is an example of runtime polymorphism because the method that gets invoked is determined at runtime based on the object's actual type (not the reference type).**

**Key Characteristics:**

- **Achieved by having the subclass provide a specific implementation of a method that is already defined in the superclass.**
- **Decided at runtime (i.e., dynamic polymorphism).**
- **It allows a subclass to define its own version of a method that is already defined in the parent class, promoting method specialization.**

**Example:**

java

CopyEdit

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

class Dog extends Animal {
    // Overridden method in subclass
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    // Overridden method in subclass
    @Override
    public void sound() {
        System.out.println("Cat meows");
    }
}
```

```
public class TestPolymorphism {
    public static void main(String[] args) {
        // Create references to the parent class (Animal) but
instantiate them with child class objects
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        // Runtime polymorphism: method that gets called is based on
the actual object type
        myDog.sound();  // Dog's sound method is called
        myCat.sound();  // Cat's sound method is called
    }
}
```

**Output:**

nginx
CopyEdit
```
Dog barks
Cat meows
```

**In this example:**

- **The `sound()` method is overridden in the `Dog` and `Cat` subclasses.**
- **Even though `myDog` and `myCat` are both declared as `Animal`, the methods from the actual object types (i.e., `Dog` and `Cat`) are called at runtime.**

#### Key Points of Runtime Polymorphism:

- **The reference type is `Animal`, but the method invoked is determined by the actual object type (`Dog` or `Cat`).**
- **This allows you to write more flexible and reusable code, where objects can be treated generically, and their specific behavior is determined at runtime.**

### Why is Polymorphism Important?

1. **Flexibility: It allows you to write more flexible and reusable code by providing a common interface for different types of objects.**

2. Extensibility: New classes can be added without modifying existing code, which is a key principle of open/closed principle (a design principle in SOLID).

3. Code Maintenance: It allows for more maintainable code because you can change the behavior of methods in subclasses without affecting the rest of the code.

4. Ease of Use: Polymorphism makes it easier to work with complex systems by allowing methods to operate on objects of different types but using the same method name.

### Real-Life Example of Polymorphism in a Spring Boot Application

Let's say you're building a Spring Boot application with different types of payment methods (e.g., CreditCardPayment, PayPalPayment, BitcoinPayment). You want to handle payments in a unified way, but each payment type has its own method of processing payments.

#### Step 1: Define an Interface

java
CopyEdit

```
public interface Payment {
    void processPayment(double amount);
}
```

#### Step 2: Implement the Interface in Different Payment Types

java
CopyEdit

```
@Service
public class CreditCardPayment implements Payment {

    @Override
    public void processPayment(double amount) {
        System.out.println("Processing Credit Card payment of
amount: " + amount);
    }
}

@Service
public class PayPalPayment implements Payment {

    @Override
    public void processPayment(double amount) {
```

```
        System.out.println("Processing PayPal payment of amount: " +
amount);
    }
}

@Service
public class BitcoinPayment implements Payment {

    @Override
    public void processPayment(double amount) {
        System.out.println("Processing Bitcoin payment of amount: "
+ amount);
    }
}
```

Step 3: Payment Service to Use Polymorphism

java

CopyEdit

```
@Service
public class PaymentService {

    @Autowired
    private Payment paymentMethod;

    public void makePayment(double amount) {
        paymentMethod.processPayment(amount); // The actual payment
method is determined at runtime
    }
}
```

Step 4: Controller Layer to Handle Payment Requests

java

CopyEdit

```
@RestController
@RequestMapping("/payment")
public class PaymentController {

    @Autowired
    private PaymentService paymentService;

    @PostMapping("/pay")
```

```
    public String makePayment(@RequestParam double amount,
@RequestParam String paymentMethod) {
        switch (paymentMethod.toLowerCase()) {
            case "creditcard":
                paymentService.makePayment(amount); // Will invoke
CreditCardPayment processPayment
                break;
            case "paypal":
                paymentService.makePayment(amount); // Will invoke
PayPalPayment processPayment
                break;
            case "bitcoin":
                paymentService.makePayment(amount); // Will invoke
BitcoinPayment processPayment
                break;
            default:
                return "Invalid payment method!";
        }
        return "Payment processed!";
    }
}
```

#### Step 5: Spring Boot Configuration

This setup uses polymorphism in the `PaymentService` class. Depending on the payment method selected, the appropriate method (`processPayment`) of the corresponding implementation class (`CreditCardPayment`, `PayPalPayment`, `BitcoinPayment`) will be invoked at runtime.

---

### Summary of Polymorphism

- Compile-time Polymorphism (Method Overloading): Methods with the same name but different parameters. Decided at compile-time.
- Runtime Polymorphism (Method Overriding): Methods in subclasses override methods in the parent class. Decided at runtime.
- Benefits: Polymorphism provides flexibility, extensibility, and code reusability. It also makes your code cleaner and more maintainable by allowing you to use a common interface for different types of objects.

**Let me know if you need more details or examples!**