Method Overriding in Java

What is Method Overriding?

- When a method is defined in both a superclass and its subclass, the subclass's version of the method replaces the superclass's version. This is called method overriding.

How Does It Work?

@Override Annotation: This tells the compiler that the method is meant to override a method from the superclass. It helps catch errors if the method doesn't match the superclass's method exactly.

Rules for Overriding:

- The method name, return type, and parameters must be the same in both the superclass and the subclass.
- You cannot override methods that are declared final or static.
- You must override abstract methods (methods without a body) from the superclass.

Note:

Constructors are not inherited and therefore cannot be overridden. However, a subclass can call its superclass's constructor using super().

Example:

```
class Animal {
  public void displayInfo() {
    System.out.println("I am an animal.");
  }
}

class Dog extends Animal {
  @Override
  public void displayInfo() {
    System.out.println("I am a dog.");
  }
}
```

```
class Main {
  public static void main(String[] args) {
    Dog d1 = new Dog();
    d1.displayInfo(); // Output: I am a dog.
  }
}
```

Super Keyword in Java

In Java, the super keyword is used in subclasses to refer to and access members (like methods, attributes, and constructors) of their Superclass / Baseclass. This is especially useful when you have overridden methods or when you need to differentiate between superclass and subclass attributes.

1. Accessing Overridden Methods

When a subclass overrides a method from its superclass, the subclass's version of the method is called. If you need to call the superclass's version of that method, you can use super.

Example:

```
class Animal {
  public void display() {
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {
  @Override
  public void display() {
    System.out.println("I am a dog");
  }

  public void printMessage() {
    display();         // Calls Dog's display()
    super.display();   // Calls Animal's display()
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```

2. Accessing Superclass Attributes

If both the superclass and subclass have attributes with the same name, super keyword helps you access the superclass's version of that attribute.

Example:

```
class Animal {
  protected String type = "animal";
}

class Dog extends Animal {
  public String type = "mammal";

  public void printType() {
    System.out.println("I am a " + type);        // Refers to Dog's type
    System.out.println("I am an " + super.type); // Refers to Animal's type
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printType();
  }
}
```

3. Calling Superclass Constructors

When creating a subclass object, the superclass constructor is called first. You can explicitly call a specific constructor from the superclass using super(). This is necessary if the superclass has a parameterized constructor.

Example:

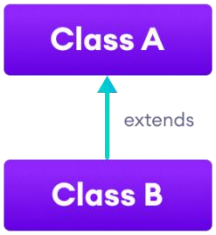
```
class Animal {
  Animal() {
    System.out.println("I am an animal");
  }

  Animal(String type) {
    System.out.println("Type: " + type);
  }
}

class Dog extends Animal {
  Dog() {
    super("Animal"); // Calls Animal's parameterized constructor
    System.out.println("I am a dog");
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}
```

Java Inheritance Types

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

1. Single Inheritance

In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes, it is also known as simple inheritance. In the below figure, 'A' is a parent class and 'B' is a child class. The class 'B' inherits all the properties of the class 'A'.

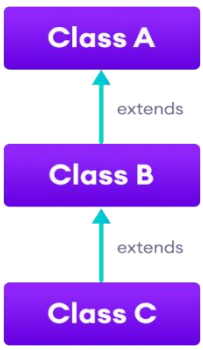


Example:

```
// Superclass
class Vehicle {
    // Method in superclass
    void start() {
        System.out.println("Vehicle is starting.");
    }
}
```

```
// Subclass
class Car extends Vehicle {
    // Additional method in subclass
    void drive() {
        System.out.println("Car is driving.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an object of the subclass
        Car myCar = new Car();

        // Call method inherited from superclass
        myCar.start();

        // Call method of subclass
        myCar.drive();
    }
}
```

2. Multilevel Inheritance: In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below figure, class A serves



as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.

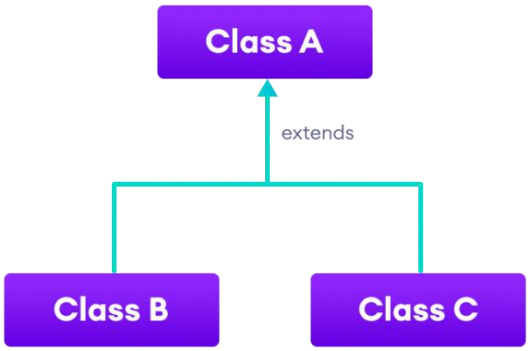
Example:

```
// Superclass
class Animal {
    // Method in superclass
    void eat() {
        System.out.println("Animal is eating.");
    }
}

// Subclass inheriting from Animal
class Mammal extends Animal {
    // Additional method in subclass
    void breathe() {
        System.out.println("Mammal is breathing.");
    }
}

// Subclass inheriting from Mammal
class Dog extends Mammal {
    // Additional method in subclass
    void bark() {
        System.out.println("Dog is barking.");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Create an object of the Dog class
        Dog myDog = new Dog();

        // Call method inherited from Animal class
        myDog.eat();

        // Call method inherited from Mammal class
        myDog.breathe();

        // Call method of Dog class
        myDog.bark();
    }
}
```

3. Hierarchical Inheritance

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below figure, class A serves as a base class for the derived classes B, and C.



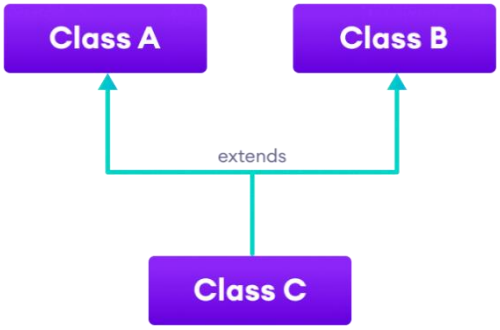
Example:

```
// Superclass
class Animal {
    // Method in superclass
    void eat() {
        System.out.println("Animal is eating.");
    }
}

// Subclass inheriting from Animal
class Dog extends Animal {
    // Additional method in subclass
    void bark() {
        System.out.println("Dog is barking.");
    }
}

// Another subclass inheriting from Animal
class Cat extends Animal {
    // Additional method in subclass
    void meow() {
        System.out.println("Cat is meowing.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an object of the Dog class
        Dog myDog = new Dog();
```

```
    // Create an object of the Cat class
        Cat myCat = new Cat();

        // Call method inherited from Animal class
        myDog.eat();
        // Call method of Dog class
        myDog.bark();

        // Call method inherited from Animal class
        myCat.eat();
        // Call method of Cat class
        myCat.meow();
    }
}
```

4. Multiple Inheritance

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. **Note:** Java does not support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the figure below, Class C is derived from interfaces A and B.



Example:

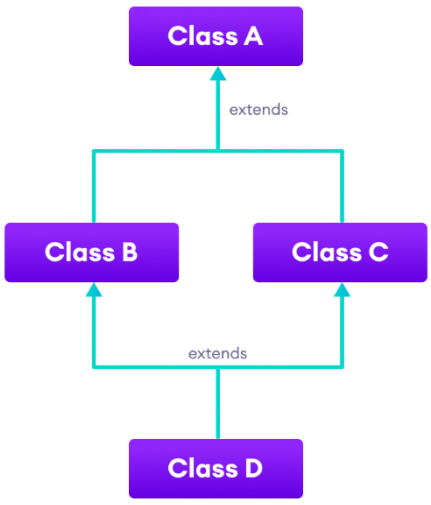
```
// First interface
interface Animal {
    void eat();
}

// Second interface
interface Pet {
    void play();
}

// Class implementing both interfaces
class Dog implements Animal, Pet {
    // Implementation of eat method from Animal interface
    public void eat() {
        System.out.println("Dog is eating.");
    }

    // Implementation of play method from Pet interface
    public void play() {
        System.out.println("Dog is playing.");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Create an object of the Dog class
        Dog myDog = new Dog();

        // Call methods from both interfaces
        myDog.eat();  // From Animal interface
        myDog.play(); // From Pet interface
    }
}
```

5. Hybrid Inheritance

Hybrid inheritance is a mix of two or more types of inheritance. Since Java doesn't support multiple inheritance with classes, hybrid inheritance that involves multiple inheritance is not possible with classes. However, you can achieve hybrid inheritance using interfaces if you need to involve multiple inheritance.

Importantly, hybrid inheritance doesn’t always need multiple inheritance. You can achieve it by combining other types of inheritance, such as multilevel inheritance and hierarchical inheritance, or hierarchical and single inheritance. Therefore, it is possible to implement hybrid inheritance using only classes, without needing multiple inheritance.



Example:

```
// Interface
interface Animal {
    void eat();
}

// Another Interface
interface Pet {
    void play();
}

// Base class
class Mammal {
    void breathe() {
        System.out.println("Mammal is breathing.");
    }
}

// Class that implements multiple interfaces and extends another class
class Dog extends Mammal implements Animal, Pet {
    // Implementation of eat method from Animal interface
    public void eat() {
        System.out.println("Dog is eating.");
    }

    // Implementation of play method from Pet interface
    public void play() {
        System.out.println("Dog is playing.");
    }

    // Additional method in Dog class
    void bark() {
        System.out.println("Dog is barking.");
    }
}
```

```
// Another subclass that extends the base class Mammal
class Cat extends Mammal {
    // Implementation of eat method from Animal interface
    void meow() {
        System.out.println("Cat is meowing.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an object of the Dog class
        Dog myDog = new Dog();

        // Create an object of the Cat class
        Cat myCat = new Cat();

        // Call methods from the Dog class
        myDog.eat();   // From Animal interface
        myDog.play();  // From Pet interface
        myDog.bark();  // From Dog class
        myDog.breathe(); // From Mammal class

        // Call methods from the Cat class
        myCat.meow();  // Specific to Cat class
        myCat.breathe(); // From Mammal class
    }
}
```