Method Overriding in Java

- **Method Overriding** happens when a **child class** provides its own implementation of a method that is already defined in the **parent class**.
- The **method signature** (name + parameters) must be **same**.
- Return type should be the same (or covariant in case of objects).

It is used to achieve **Runtime Polymorphism**.

Rules for Method Overriding

1. Method name and parameters must be the **same**.
2. Return type must be the **same** (or covariant).
3. Access modifier of child's method **cannot be more restrictive** than parent's.
    - Example: if parent method is `public`, child cannot make it `private`.
4. Only **inherited methods** can be overridden.
5. **Constructors cannot be overridden**.
6. Use `@Override` annotation for clarity.

Example

```
class Employee {
    String name;
    double salary;

    Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    void displayInfo() {
        System.out.println("Employee: " + name + ", Salary: " + salary);
    }
}

class Manager extends Employee {
    String department;

    Manager(String name, double salary, String department) {
        super(name, salary);
        this.department = department;
    }

    // Overriding the displayInfo method
    @Override
    void displayInfo() {
        System.out.println("Manager: " + name + ", Salary: " + salary + ",
Department: " + department);
    }
}

public class Main {
    public static void main(String[] args) {
        Employee e = new Employee("Neeraj", 50000);
        Manager m = new Manager("Rahul", 70000, "IT");

        e.displayInfo();  // Parent class method
        m.displayInfo();  // Overridden method in Manager class
    }
```

```
}
```

Output:

```
Employee: Neeraj, Salary: 50000.0
Manager: Rahul, Salary: 70000.0, Department: IT
```

Key Point: super in Overriding

Child can still call **parent class method** using `super`:

```
@Override
void displayInfo() {
    super.displayInfo();  // calls parent method
    System.out.println("Department: " + department);
}
```

Dynamic Method Dispatch in Java

- **Dynamic Method Dispatch** (also called **runtime polymorphism**) is the process in which:
- A **method call** to an **overridden method** is resolved **at runtime**, not at compile time.
- It allows Java to decide **which version of method to execute (parent's or child's)** depending on the **object type** being referred.

How it Works:

- Parent class reference variable can point to **child class object**.
- Which method runs depends on **object type**, not reference type.

Note: *Compile-time checks reference type, but runtime executes actual object's method.*

Example

```
class Employee {
    String name;
    double salary;

    Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    void displayInfo() {
        System.out.println("Employee: " + name + ", Salary: " + salary);
    }
}

class Manager extends Employee {
    String department;

    Manager(String name, double salary, String department) {
        super(name, salary);
        this.department = department;
    }

    @Override
    void displayInfo() {
        System.out.println("Manager: " + name + ", Salary: " + salary + ",
Department: " + department);
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Employee emp;              // reference of parent
        emp = new Employee("Neeraj", 40000);
        emp.displayInfo();         // Employee version

        emp = new Manager("Rahul", 70000, "IT");
        emp.displayInfo();         // Manager version (child class method)
    }
}
```

Output:

```
Employee: Neeraj, Salary: 40000.0
Manager: Rahul, Salary: 70000.0, Department: IT
```

Key Points

a) Achieved through **method overriding**.
b) Object type decides **which method** will execute.
c) Used for **runtime polymorphism**.
d) Reference type = decides what methods are accessible at compile time.
e) Object type = decides which overridden method runs at runtime.

Abstract Class and Abstract Methods in Java

1. Abstract Class

- An **abstract class** in Java is declared using the **`abstract` keyword**.
- You **cannot create an object** of an abstract class directly.
- It works like a **blueprint** and can contain both:
    - **Abstract methods** (without body)
    - **Normal methods** (with body)

2. Abstract Method

- An **abstract method** only provides **declaration**, not implementation.
- The **implementation must be provided by a subclass**.
- Abstract methods **cannot have a body**.
- If a class contains an abstract method, that class must also be declared as **abstract**.

3. Key Points

- An abstract class **can have a constructor**.
- It can contain **both abstract and non-abstract methods**.
- If a subclass inherits an abstract class, it **must implement all abstract methods** (unless the subclass is also abstract).

Example:

```
// Abstract class
abstract class Employee {
    String name;
    double salary;
```

```
    // Constructor
    Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    // Abstract method
    abstract void work();

    // Normal method
    void displayDetails() {
        System.out.println("Name: " + name + ", Salary: " + salary);
    }
}

// Subclass
class Manager extends Employee {
    Manager(String name, double salary) {
        super(name, salary);
    }

    // Implementing abstract method
    void work() {
        System.out.println(name + " is managing the team.");
    }
}

// Another Subclass
class Developer extends Employee {
    Developer(String name, double salary) {
        super(name, salary);
    }

    void work() {
        System.out.println(name + " is writing code.");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Employee e1 = new Manager("Rahul", 50000);
        Employee e2 = new Developer("Neha", 60000);

        e1.displayDetails();
        e1.work();

        e2.displayDetails();
        e2.work();
    }
}
```

Output

```
Name: Rahul, Salary: 50000.0
Rahul is managing the team.
Name: Neha, Salary: 60000.0
Neha is writing code.
```

Interface in Java

1. What is an Interface?

- An **interface** in Java is a collection of **abstract methods** (methods without body).
- It is used to **achieve abstraction** and **multiple inheritance** in Java.
- By default, all variables inside an interface are:
  - `public static final` (constants)
- All methods are:
  - `public abstract` (implicitly, even if you don't write them).

Syntax:

```
interface InterfaceName {
    // Abstract method
    void method1();

    // Variables (constants)
    int VALUE = 100;
}
```

2. Implementing an Interface

- A class **uses the `implements` keyword** to use an interface.
- The class must provide **implementation for all methods** in the interface.

Example:

```
interface Animal {
    void sound();  // abstract method
    void sleep();
}

class Dog implements Animal {
    public void sound() {
        System.out.println("Dog barks");
    }
    public void sleep() {
        System.out.println("Dog sleeps at night");
    }
}

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

Output:

```
Dog barks
Dog sleeps at night
```

3. Multiple Inheritance with Interfaces

- A class can implement **multiple interfaces** (unlike classes, where multiple inheritance is not allowed).

Example:

```
interface Printable {
    void print();
}
interface Showable {
    void show();
}

class Demo implements Printable, Showable {
    public void print() {
        System.out.println("Printing...");
    }
    public void show() {
        System.out.println("Showing...");
    }
}

public class Main {
    public static void main(String[] args) {
        Demo obj = new Demo();
        obj.print();
        obj.show();
    }
}
```

Output:

```
Printing...
Showing...
```

4. Default and Static Methods in Interface (Java 8+)

- **Default methods**: Interfaces can have methods with a body using the `default` keyword.
- **Static methods**: Interfaces can also contain static methods.

Example:

```
interface Vehicle {
    void drive();

    default void fuel() {
        System.out.println("This vehicle uses fuel");
         price();
    }

    static void service() {
        System.out.println("Vehicle needs servicing");
    }
    private void price() {
        System.out.println("Vehicle price is:4000");
    }

}

class Car implements Vehicle {
    public void drive() {
        System.out.println("Car is driving");
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        Car c = new Car();
        c.drive();
        c.fuel();       // default method
        Vehicle.service(); // static method
    }
}
```

Output:

```
Car is driving
This vehicle uses fuel
Vehicle needs servicing
```

5. Key Differences: Abstract Class vs Interface

Feature	Abstract Class	Interface
Keyword	`abstract`	`interface`
Methods	Abstract + Normal	Only Abstract (till Java 7), Abstract + Default + Static (Java 8+)
Variables	Any type	Always `public static final`
Multiple Inheritance	Not supported	Supported
Constructor	Allowed	Not allowed

Interface Inheritance and Polymorphism in Java

1. Interface Inheritance

- In Java, an **interface can extend another interface** using the **`extends` keyword**.
- A class that implements the child interface must provide implementations for **all methods** (from parent + child).
- Unlike classes, interfaces support **multiple inheritance**.

Example:

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

interface Dog extends Animal {
    void bark();
}

class PetDog implements Dog {
    public void eat() {
        System.out.println("Dog eats food");
    }
    public void bark() {
        System.out.println("Dog barks loudly");
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        PetDog d = new PetDog();
        d.eat();
        d.bark();
    }
}
```

Output:

```
Dog eats food
Dog barks loudly
```

Here, `Dog` interface inherits from `Animal`.
`PetDog` must implement **both** `eat()` and `bark()`.

2. Polymorphism with Interfaces

- **Polymorphism** means "**many forms**".
- In Java, it allows **one reference type** (like an interface) to point to **different objects**.
- Interfaces are commonly used to achieve **runtime polymorphism**.

Example 1:

```
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 Main {
    public static void main(String[] args) {
        // Interface reference, multiple implementations
        Shape s1 = new Circle();
        Shape s2 = new Rectangle();

        s1.draw(); // Circle's implementation
        s2.draw(); // Rectangle's implementation
    }
}
```

Output:

```
Drawing a Circle
Drawing a Rectangle
```

Here, `Shape` interface reference is used to call `draw()` on different objects.
The actual method executed depends on the **object type at runtime** → this is **runtime polymorphism**.

Example 2:

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

// Second interface
interface Color {
    void fillColor();
}

// Third interface
interface Resizable {
    void resize();
}

// One class implements all interfaces
class Circle implements Shape, Color, Resizable {
    public void draw() {
        System.out.println("Drawing a Circle");
    }
    public void fillColor() {
        System.out.println("Filling Circle with Red color");
    }
    public void resize() {
        System.out.println("Resizing the Circle");
    }
}

public class Main {
    public static void main(String[] args) {
        // Interface references for polymorphism
        Shape s = new Circle();     // reference of Shape
        Color c = new Circle();     // reference of Color
        Resizable r = new Circle(); // reference of Resizable

        // Each reference calls its own method
        s.draw();
        c.fillColor();
        r.resize();
    }
}
```

Output:

```
Drawing a Circle
Filling Circle with Red color
Resizing the Circle
```