Zoho Schools for Graduate Studies

Notes

DAY - 5(23-07-2025)

Session -1

Single Inheritance:

Definition:

Single Inheritance in Java is when a class (child/subclass) inherits the properties and behaviors (methods/fields) of **one parent (superclass)**. It promotes **code reusability** and establishes a parent-child relationship between classes.

What we can do:

1. Inherit **fields and methods** of the parent class.
2. Override the parent class methods if needed.
3. Add **new methods** in the child class.
4. Use super keyword to access parent class members.

Example Code:

```
// Parent Class
class Animal {
    String color = "Brown";
    boolean domestic = true;
}

// Child Class 1
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks: Woof Woof!");
    }
}
```

```
// Child Class 2
class Cat extends Animal {
    void meow() {
        System.out.println("Cat meows: Meow Meow!");
    }
}

// Main Class
public class TestInheritance {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.bark();
        Cat cat = new Cat();
        cat.meow();
    }
}
```

Multi Level Inheritance:

Definition:

Multilevel Inheritance is a type of inheritance where a class is derived from a class, which is also derived from another class. It forms a **chain of inheritance.**

Concepts:

1. The **base class (Animal)** has general properties.
2. The **intermediate class (Dog)** inherits from Animal and adds specific behavior.
3. The **derived class (Puppy)** inherits from Dog and adds more specialized behavior.

4. Each subclass can access **all non-private** properties and methods of its parent and grandparent classes.

Example Code:

```
// Base Class
class Animal {
    String color = "Brown";
    boolean domestic = true;
}

// Derived Class 1
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks: Woof Woof!");
    }
}

// Derived Class 2 (Multilevel Inheritance)
class Puppy extends Dog {
    void weep() {
        System.out.println("Puppy weeps: Whimper Whimper!");
    }
}
```

```
// Main Class
public class TestMultilevelInheritance {
    public static void main(String[] args) {
        Puppy pup = new Puppy();
        // Accessing all methods from parent, grandparent
        pup.bark();          // From Dog
        pup.weep();          // Own method
    }
}
```

Heirarchical Inheritance:

Definition:

Hierarchical Inheritance occurs when **multiple subclasses inherit from a single parent class**.
Each subclass shares the common properties/methods of the parent but can also have its **own unique behavior**.

Class Structure:

```
      Animal
      /   \
    Dog    Cat
```

Concepts:

1. Use shared logic (e.g., color, domestic) in parent class.
2. Have individual behavior in each subclass (bark(), meow()).
3. Create multiple child classes from one base class

4. One subclass (like Dog) **cannot access another subclass's** (Cat) unique methods.
5. Cannot override **private** members of the parent.

6. Multiple inheritance of classes is not allowed (only one parent class per subclass).

Multiple Inheritance:

Why Java Does NOT Support Multiple Inheritance with Classes:

The main reason is to **avoid ambiguity** and **confusion in the Diamond Problem.**

Diamond Problem:

Suppose two parent classes have the same method, and a child class inherits from both — **which version should the child use?**

Example Code:

```
class A {
    void show() {
        System.out.println("A's show");
    }
}

class B {
    void show() {
        System.out.println("B's show");
    }
}

// This will cause error in Java
class C extends A, B {       // ❌ Invalid
    // Ambiguity: Which show() to use?
}
```

- Interfaces allow Java to achieve multiple inheritance **safely**, as they only contain method **signatures** (no logic), avoiding ambiguity.

Session -2

Overiding:

Definition:

Method Overriding in Java is a feature that allows a subclass to provide a **specific implementation** of a method that is already defined in its superclass.
The overridden method in the subclass must have:

1. The **same method name**
2. The **same parameter list**
3. The **same or covariant return type**

Rule	Description
Same signature	Method name and parameters must match exactly.
Access specifier	Can be more visible but not less (e.g., protected → public is OK).
Non-static	Only **instance methods** can be overridden (not static methods).
Use of @Override	Helps the compiler catch mistakes.
Final methods	Cannot be overridden.
Private methods	Not inherited, so cannot be overridden.

Example Code:

```
// Superclass
class RBI {
    double getInterestRate() {
        return 5.0;  // Default interest rate
    }
}

// Subclass 1
class SBI extends RBI {
    @Override
    double getInterestRate() {
        return 6.5;  // SBI-specific interest rate
    }
}

// Subclass 2
class HDFC extends RBI {
    @Override
    double getInterestRate() {
        return 7.0;  // HDFC-specific interest rate
    }
}
// Main Class
public class TestOverriding {
    public static void main(String[] args) {
        RBI bank1 = new SBI();   // RBI reference to SBI object
        RBI bank2 = new HDFC();  // RBI reference to HDFC object

        System.out.println("SBI Interest Rate: " +
bank1.getInterestRate());
        System.out.println("HDFC Interest Rate: " +
bank2.getInterestRate());
    }
}
```

How It Works (Behind the Scenes):

1. The reference type is **RBI**, but the actual object is **SBI or HDFC.**
2. At **runtime**, Java dynamically decides which method to call based on the object, not the reference type — this is **runtime polymorphism (dynamic dispatch).**
3. Even though bank1 and bank2 are of type RBI, the overridden methods in SBI and HDFC are called.

Overloading :

Definition:

Method Overloading is a feature in Java where **two or more methods** in the **same class** have the **same name** but **different parameter lists** (type, number, or order of parameters).
It is a type of **compile-time polymorphism**, meaning the method to execute is decided during **compilation**, not runtime.

Why Use Overloading?

- To perform the **same logical operation** in **different ways.**
- Increases **readability** and **reusability** of code.
- Simplifies method calls by handling **various input types.**

Example Code :

```
class ArithmeticOperations {

    // 1. Add two integers
    int add(int a, int b) {
        return a + b;
    }

    // 2. Add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }
```

```
 // 3. Add two doubles
    double add(double a, double b) {
        return a + b;
    }

// 4. Multiply two integers
    int multiply(int a, int b) {
        return a * b;
    }

    // 5. Multiply three integers
    int multiply(int a, int b, int c) {
        return a * b * c;
    }
}

public class TestOverloading {
    public static void main(String[] args) {
        ArithmeticOperations op = new ArithmeticOperations();

        System.out.println("Add 2 int: " + op.add(5, 10));
        System.out.println("Add 3 int: " + op.add(1, 2, 3));
        System.out.println("Add 2 double: " + op.add(4.5, 3.2));

        System.out.println("Multiply 2 int: " + op.multiply(3, 4));
        System.out.println("Multiply 3 int: " + op.multiply(2, 3, 4));
    }
}
```

Rule	**Explanation**
Return type ignored	Only parameters are considered for overloading.
Cannot overload by changing only return type	E.g., int add(int, int) vs double add(int, int) is invalid.
Works at compile-time	This is why it's called **compile-time polymorphism**.

Generics :

Definition :

Generics allow you to write **a single class, interface, or method** that can operate on **different data types** without sacrificing **type safety**.

Introduced in **Java 5**, Generics provide **compile-time type checking** and eliminate the need for **type casting**.

Benefit	Explanation
🔒 **Type Safety**	Errors are caught during compile time, not at runtime.
📦 **Code Reusability**	Write one class or method that works for all types.
🚫 **No Casting Needed**	Avoids messy casting logic like (Integer) obj.
✅ **Cleaner Code**	Reduces redundancy, improves readability.

Where Generics Are Used:

- **Collections Framework** (like ArrayList<String>)
- **User-defined classes** and **methods**
- **Wrapper classes** or **utility classes**

Example: Generic Class for Arithmetic Operation (Addition) :

```
// Generic Arithmetic Class
class Adder<T extends Number> {
    T num1, num2;

    Adder(T num1, T num2) {
        this.num1 = num1;
        this.num2 = num2;
    }

 double add() {
        // Convert both numbers to double for addition
        return num1.doubleValue() + num2.doubleValue();
    }
}


// Main Class
public class TestGenericAddition {
    public static void main(String[] args) {
        Adder<Integer> intAdder = new Adder<>(10, 20);
        System.out.println("Integer Addition: " + intAdder.add());

        Adder<Double> doubleAdder = new Adder<>(5.5, 6.3);
        System.out.println("Double Addition: " + doubleAdder.add());

        Adder<Float> floatAdder = new Adder<>(3.2f, 4.8f);
        System.out.println("Float Addition: " + floatAdder.add());
    }
}
```