Java Unit - 3

Inheritance and Polymorphism : Inheritance, Method overriding, super keyword, Object class and overriding toString() and equals() method, Using super and final keywords, instanceof operator

Abstract Class and Interface : Abstract method and abstract class, Interfaces, static and default methods.

Inheritance and Polymorphism

Inheritance

Inheritance is a fundamental concept of Object-Oriented Programming (OOP) in Java that allows **a class to inherit properties (fields) and behavior (methods) from another class.** It helps in **code reusability**, **method overriding**, and creating a hierarchical classification of objects.

The class that **inherits** from another class is called the **subclass (child class)**, and the class from which it inherits is called the **superclass (parent class)**.

Key Benefits of Inheritance

- **Code Reusability**: Common properties and methods can be defined in a parent class and reused in child classes.
- **Method Overriding**: Allows a subclass to provide a specific implementation of a method already defined in the parent class.
- **Polymorphism Support**: Helps in achieving dynamic method dispatch (runtime polymorphism).
- **Better Organization**: Creates a logical hierarchy of classes in a program.

Types of Inheritance in Java

Java supports different types of inheritance:

1. **Single Inheritance**
2. **Multilevel Inheritance**
3. **Hierarchical Inheritance**
4. **Multiple Inheritance using Interfaces**

Note: Java does not support **Multiple Inheritance** using classes to avoid ambiguity/diamond problem , but it can be achieved using **interfaces**.

---

1. Single Inheritance

Definition: In Single Inheritance, a child class inherits from a single parent class.

Example:

```
class Vehicle {                                          Ummed Singh
    String brand = "Toyota";                                  ByteXL
    void honk() {
        System.out.println("Vehicle is honking...");
    }
}

// Child class (Sub class)
class Car extends Vehicle {
    int speed = 100;
    void display() {
        System.out.println("Brand: " + brand);
        System.out.println("Speed: " + speed + " km/h");
    }
}

// Main class
public class SingleInheritanceExample {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.display();
        myCar.honk();
    }
}
Output:
Brand: Toyota
Speed: 100 km/h
Vehicle is honking...
```

Real-World Use Case:

💡 **Banking System (User Account Inheritance)**

- **Parent Class**: `BankAccount` (common properties like balance, account number)
- **Child Class**: `SavingsAccount`, `CurrentAccount`

```
class BankAccount {
    String accountHolder;
    double balance;

    BankAccount(String accountHolder, double balance) {
        this.accountHolder = accountHolder;
        this.balance = balance;
    }

    void deposit(double amount) {
        balance += amount;
        System.out.println(amount + " deposited. New balance: " +
balance);
    }
}
// Child class inheriting from BankAccount
class SavingsAccount extends BankAccount {
    SavingsAccount(String accountHolder, double balance) {
        super(accountHolder, balance);
    }
    void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println(amount + " withdrawn. Remaining
balance: " + balance);
        } else {
            System.out.println("Insufficient balance.");
        }
    }
}

// Main class to test the implementation
public class BankingSystem {
    public static void main(String[] args) {
        SavingsAccount myAccount = new SavingsAccount("Ummed
Singh", 500.0);
        myAccount.deposit(200);
        myAccount.withdraw(100);
```

```
        myAccount.withdraw(700);
    }
}
```

Problem Without Inheritance

If every account type (`SavingsAccount`, `CurrentAccount`) defines balance and deposit separately, it leads to duplicate code.

Solution Through Inheritance

By making `BankAccount` a superclass, all account types can inherit common properties, avoiding redundancy.

---

2. Multilevel Inheritance

Definition: In Multilevel Inheritance, a class inherits from another class, which in turn is inherited by another class.

Example:

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}
// Derived class
class Mammal extends Animal {
    void walk() {
        System.out.println("Mammals can walk.");
    }
}
// Derived from Mammal
class Human extends Mammal {
    void speak() {
        System.out.println("Humans can speak.");
    }
}
// Main class
public class MultilevelInheritanceExample {
    public static void main(String[] args) {
```

```
        Human human = new Human();
        human.eat();
        human.walk();
        human.speak();
    }
}

Output:
This animal eats food.
Mammals can walk.
Humans can speak.
```

Real-World Use Case:

💡 **Educational Management System**

- **Parent Class**: `Person` (common properties like name, age)
- **Child Class**: `Student` (adds roll number)
- **Grandchild Class**: `EngineeringStudent` (specializes further)

Problem Without Inheritance

Redundant fields for each role like `name`, `age`, `gender`.

Solution Through Inheritance

By inheriting `Person`, `Student`, and `EngineeringStudent`, we create a structured hierarchy.

```
class Person {

    String name;

    Person(String name) {

        this.name = name;

    }

    void displayPersonInfo() {

        System.out.println("Name: " + name);
```

```
    }
}
// Child class inheriting from Person
class Student extends Person {
    int rollNumber;
    Student(String name, int rollNumber) {
        super(name);
        this.rollNumber = rollNumber;
    }
    void displayStudentInfo() {
        displayPersonInfo(); // Calling parent class method
        System.out.println("Roll Number: " + rollNumber);
    }
}
// Grandchild class inheriting from Student
class EngineeringStudent extends Student {
    String specialization;
    EngineeringStudent(String name, int rollNumber, String
specialization) {
        super(name, rollNumber);
        this.specialization = specialization;
    }
    void displayEngineeringStudentInfo() {
        displayStudentInfo(); // Calling student class method
```

```
        System.out.println("Specialization: " + specialization);

    }

}

// Main class to test the hierarchy

public class EducationSystem {

    public static void main(String[] args) {

        EngineeringStudent engStudent = new
EngineeringStudent("Ummed Singh", 1001, "Computer Science");

        engStudent.displayEngineeringStudentInfo();

    }

}
```

---

3. Hierarchical Inheritance

Definition: In Hierarchical Inheritance, multiple child classes inherit from a single parent class.

Example:

```
// Parent class
class Shape {
    void area() {
        System.out.println("Calculating area...");
    }
}

// Child class 1
class Circle extends Shape {
    double radius = 5;
    void area() {
        System.out.println("Area of Circle: " + (3.14 * radius *
radius));
    }
```

```
}

// Child class 2
class Rectangle extends Shape {
    double length = 4, width = 6;
    void area() {
        System.out.println("Area of Rectangle: " + (length *
width));
    }
}

// Main class
public class HierarchicalInheritanceExample {
    public static void main(String[] args) {
        Circle c = new Circle();
        c.area();
        Rectangle r = new Rectangle();
        r.area();
    }
}
```

Real-World Use Case:

💡 **E-Commerce Payment System**

- **Parent Class**: `Payment` (common methods like `processPayment()`)
- **Child Classes**: `CreditCardPayment`, `UPIPayment`, `WalletPayment`

Problem Without Inheritance

Each payment method must implement `processPayment()` separately.

Solution Through Inheritance

Create a `Payment` base class to enforce consistency.

```
abstract class Payment {

    double amount;

    Payment(double amount) {
```

```
        this.amount = amount;
    }
    abstract void processPayment();
}
// Child class for Credit Card Payment
class CreditCardPayment extends Payment {
    String cardNumber;
    CreditCardPayment(double amount, String cardNumber) {
        super(amount);
        this.cardNumber = cardNumber;
    }
    @Override
    void processPayment() {
        System.out.println("Processing Credit Card Payment of $"
+ amount + " using card " + cardNumber);
    }
}
// Child class for UPI Payment
class UPIPayment extends Payment {
    String upiId;
    UPIPayment(double amount, String upiId) {
        super(amount);
        this.upiId = upiId;
    }
```

```
    @Override
    void processPayment() {
        System.out.println("Processing UPI Payment of $" + amount
+ " using UPI ID " + upiId);
    }
}
// Main class to test the implementation
public class ECommercePaymentSystem {
    public static void main(String[] args) {
        Payment creditCard = new CreditCardPayment(250.75,
"1234-5678-9876-5432");
        creditCard.processPayment();
        Payment upi = new UPIPayment(100.50, "user@upi");
        upi.processPayment();
    }
}
```

---

4. Multiple Inheritance using Interfaces

Java does **not support multiple inheritance** with classes to avoid ambiguity but allows it with **interfaces**.

Example:

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

interface Showable {
```

```
    void show();
}

// Multiple inheritance through interfaces
class Document implements Printable, Showable {
    public void print() {
        System.out.println("Printing document...");
    }

    public void show() {
        System.out.println("Showing document...");
    }
}

public class MultipleInheritanceExample {
    public static void main(String[] args) {
        Document doc = new Document();
        doc.print();
        doc.show();
    }
}
```

Real-World Use Case:

💡 **Messaging System**

- **Interface 1**: `Sendable` (defines `send()`)
- **Interface 2**: `Receivable` (defines `receive()`)
- **Class**: `Email` (implements both interfaces)

Problem Without Inheritance

Emails and SMS need to implement `send()` and `receive()` separately.

Solution Through Inheritance

A single class implements both interfaces for **code reusability**.

```
// Interface for sending messages

interface Sendable {

    void send(String message);
```

```
}

// Interface for receiving messages

interface Receivable {

   void receive();

}

// Class implementing both interfaces

class Email implements Sendable, Receivable {

   private String inboxMessage;

   @Override

   public void send(String message) {

       System.out.println("Sending Email: " + message);

       inboxMessage = message; // Simulating sending an email

   }

   @Override

   public void receive() {

       if (inboxMessage != null) {

           System.out.println("Received Email: " +
inboxMessage);

       } else {

           System.out.println("No new emails.");

       }

   }

}

// Main class to test the implementation
```

```java
public class MessagingSystem {

   public static void main(String[] args) {

       Email email = new Email();

       email.send("Hello, this is a test email.");

       email.receive();

   }

}
```

Diamond Problem in Java and Its Solution

The **diamond problem** occurs in programming languages that support **multiple inheritance**, leading to ambiguity when a class inherits from multiple classes that have methods with the same name.

Java **avoids this problem** by **not allowing multiple inheritance with classes** but allows it with **interfaces**.

Diamond Problem in Multiple Inheritance (Not Allowed in Java)

Problem Example (If Java Allowed Multiple Inheritance)

```java
// Parent Class 1
class A {
   void show() {
       System.out.println("A's show method");
   }
}

// Parent Class 2
class B {
```

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

// Child Class trying to inherit both A and B (Not Allowed in
Java)
class C extends A, B { // ❌ ERROR: Java does not support
multiple class inheritance
    public static void main(String[] args) {
        C obj = new C();
        obj.show(); // ❓ Ambiguity: Which show() method to call?
A's or B's?
    }
}
```

Problem:

- If Java allowed multiple inheritance with classes, class `C` would **inherit the `show()` method from both `A` and `B`**, causing **ambiguity**.

Solution: Use Interfaces in Java

Java solves the diamond problem by **allowing multiple inheritance using interfaces**, where ambiguity is resolved explicitly.

Example Using Interfaces (Allowed in Java)

```
// Interface 1
interface A {
    default void show() {
        System.out.println("A's show method");
    }
}

// Interface 2
interface B {
    default void show() {
        System.out.println("B's show method");
```

```
    }
}

// Class implementing both interfaces
class C implements A, B {
    // Overriding show() method to resolve ambiguity
    @Override
    public void show() {
        System.out.println("Resolving ambiguity...");
        A.super.show(); // Explicitly calling A's show()
        B.super.show(); // Explicitly calling B's show()
    }

    public static void main(String[] args) {
        C obj = new C();
        obj.show();
    }
}

Output:
Resolving ambiguity...
A's show method
B's show method
```

How This Works in Java

1. **Interfaces allow default methods** (Java 8+) to define behavior.
2. If multiple interfaces have the **same method**, Java forces the child class to **override it** and specify which version to use.
3. The child class can use `A.super.methodName()` or `B.super.methodName()` to resolve ambiguity explicitly.

---

1. What is Polymorphism?

Polymorphism is a core concept in Object-Oriented Programming (OOP) that **allows a single interface to be used for different types.** It enables code reusability and flexibility, making it easier to manage and extend.

Types of Polymorphism in Java:

1. **Compile-Time Polymorphism/static type (Method Overloading):** Method Overloading allows multiple methods in the same class with the same name but different parameters. The compiler determines which method to invoke based on the method signature.
2. **Run-Time Polymorphism/dynamic type (Method Overriding)**

Run-Time Polymorphism (Method Overriding)

Method Overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This is achieved using **dynamic method dispatch**, where the method call is resolved at runtime.

Example of Method Overriding in a Real-World Software (Ride Booking App like Uber):

```
// Parent class
class Ride {
    void calculateFare(int distance) {
        System.out.println("Ride fare for " + distance + " km is
calculated.");
    }
}
// Child class for Economy ride
class EconomyRide extends Ride {
    @Override
    void calculateFare(int distance) {
        System.out.println("Economy ride fare for " + distance +
" km: $" + (distance * 2));
    }
}

// Child class for Luxury ride
```

```java
class LuxuryRide extends Ride {
    @Override
    void calculateFare(int distance) {
        System.out.println("Luxury ride fare for " + distance + " km: $" + (distance * 5));

    }
}

// Main application class
public class RideBookingAppOverriding {
    public static void main(String[] args) {
        Ride myRide; // Reference variable of parent class

        myRide = new EconomyRide();
        myRide.calculateFare(10); // Calls EconomyRide's method

        myRide = new LuxuryRide();
        myRide.calculateFare(10); // Calls LuxuryRide's method
    }
}
```

Advantages of Polymorphism in Java

✅ **Code Reusability** – We can write a common interface with multiple implementations.
✅ **Scalability** – Easy to add new features (new ride types in the Uber example).
✅ **Maintainability** – Clean and structured code with better separation of concerns.

`super` Keyword

The `super` keyword in Java is used to refer to the **parent class** (superclass) of a subclass. It is primarily used for the following purposes:

1. **Access Parent Class Methods** (Method Overriding): When a subclass overrides a method from its parent class, the overridden method can still be accessed using `super.methodName()`.
2. **Call Parent Class Constructor:** When a subclass object is created, the parent class constructor runs before the subclass constructor or It must be the first statement in the subclass constructor. The **super()** keyword allows us to explicitly call a specific parent constructor.
3. **Access Parent Class Variables** (When Hidden by Subclass): If a subclass defines a variable with the same name as its parent class, it **hides** the parent's variable. The `super` keyword allows access to the parent class variable.

   `super` is commonly used to eliminate ambiguity between parent and child class members when they have the same name.

Example:

```
class ParentSuper {
   String name = "Parent";

   ParentSuper(){
       System.out.println("Executed the ParentClass
constructor");
   }
   void display() {
       System.out.println("This is the parent class.");
   }
}
class ChildSuper extends ParentSuper {
   String name = "Child";

   ChildSuper(){
       super();

   }

   void display() {


       System.out.println("This is the child class.");
   }
   void show() {
```

```
        System.out.println("Name in Child class: " + name);
        System.out.println("Name in Parent class: " +
super.name);
        display();
// Calls Child's display method
        super.display(); // Calls Parent's display method
    }
}
public class SuperDemo {
    public static void main(String[] args) {
        ChildSuper child = new ChildSuper();
        child.show();
    }
}
```

Introduction to Object Class

In Java, the `Object` class is the **superclass of all classes**. Every class in Java **implicitly** extends the `Object` class (except when explicitly extending another class).

Key Points:

- The `Object` class is part of `java.lang` package.
- It provides basic methods that all Java objects inherit.
- It allows Java to implement key object-oriented programming concepts like **polymorphism** and **inheritance**.

---

2. Methods of Object Class

The `Object` class provides several important methods that can be overridden in subclasses for customization.

List of Important Methods in the Object Class

Method	Description

`equals(Object obj)`	Checks if two objects are equal.
`hashCode()`	Returns a hash code value for the object.
`toString()`	Returns a string representation of the object.
`clone()`	Creates a duplicate (shallow copy) of an object.
`finalize()`	Called by the garbage collector before destroying an object.
`getClass()`	Returns the runtime class of the object.
`wait()`, `notify()`, `notifyAll()`	Used in thread synchronization.

Understanding Object Class Methods with Examples

(i) equals() Method

The `equals()` method compares two objects for equality. By default, it uses the `==` operator, which checks **memory address (reference equality)**.

Example Without Overriding `equals()`

```
class Car {
   String model;

   Car(String model) {
       this.model = model;
```

```
    }
}

public class ObjectEqualsExample {
    public static void main(String[] args) {
        Car car1 = new Car("Tesla");
        Car car2 = new Car("Tesla");

        System.out.println(car1.equals(car2)); // false
(different memory locations)
    }
}
```

Overriding `equals()` for Logical Comparison

```
class Car1 {

    String model;



    Car1(String model) {

        this.model = model;

    }



    @Override

    public boolean equals(Object obj) {

        if (this == obj) return true; // Same reference

        if (obj == null || getClass() != obj.getClass()) return
false;



        Car1 car = (Car1) obj;

        return model.equals(car.model);
```

```
    }
}


public class ObjectEqualsExample1 {
    public static void main(String[] args) {
        Car1 car1 = new Car1("Tesla");
        Car1 car2 = new Car1("Tesla");

        System.out.println(car1.equals(car2)); // true (based on
model name)
    }
}
```

(ii) hashCode() Method

The `hashCode()` method returns an integer that represents the **memory address** or a **logical value**. It is important when storing objects in **hash-based collections (e.g., HashMap, HashSet)**.

Example Without Overriding `hashCode()`

```
class Car2 {
    String model;

    Car2(String model) {
        this.model = model;
    }
```

```
}


public class ObjectHashCodeExample {

   public static void main(String[] args) {

       Car2 car1 = new Car2("BMW");

       Car2 car2 = new Car2("BMW");


       System.out.println(car1.hashCode()); // Different hash
codes

       System.out.println(car2.hashCode()); // Different hash
codes

   }

}
```

Overriding `hashCode()` for Consistent Hashing

```
class Car3 {

   String model;


   Car3(String model) {

       this.model = model;

   }


   @Override
```

```
    public int hashCode() {
        return model.hashCode();
    }
}

public class ObjectHashCodeExample1 {
    public static void main(String[] args) {
        Car3 car1 = new Car3("BMW");
        Car3 car2 = new Car3("BMW");

        System.out.println(car1.hashCode()); // Same hash code
        System.out.println(car2.hashCode()); // Same hash code
    }
}
```

Relation Between `equals()` and `hashCode()`

- If `equals()` returns `true`, then `hashCode()` must return the same value.
- If `equals()` returns `false`, `hashCode()` can be different.

(iii) toString() Method

The `toString()` method returns a **string representation** of an object.

Default `toString()` Implementation

```
class Car4 {
```

```
}


public class ObjectToStringExample {
    public static void main(String[] args) {
        Car4 car = new Car4();
        System.out.println(car.toString()); // Output:
Car@7d4991ad (memory reference)
    }
}
```

Overriding `toString()` for Better Output

```
class Car6 {
    String model;


    Car6(String model) {
        this.model = model;
    }


    @Override
    public String toString() {
        return "Car Model: " + model;
    }
```

```
}


public class ObjectToStringExample1 {

   public static void main(String[] args) {

       Car6 car = new Car6("Audi");

       System.out.println(car.toString()); // Output: Car Model:
Audi

   }

}
```

(iv) clone() Method (Shallow Copy)

The `clone()` method creates a **copy of an object**. It requires the **`Cloneable`** interface.

Example of `clone()`

```
class Car7 implements Cloneable {

   String model;


   Car7(String model) {

       this.model = model;

   }


   @Override

   protected Object clone() throws CloneNotSupportedException {

       return super.clone();
```

```
    }
}


public class ObjectCloneExample {
    public static void main(String[] args) throws
CloneNotSupportedException {
        Car7 car1 = new Car7("Tesla");
        Car7 car2 = (Car7) car1.clone();

        System.out.println(car1.model); // Tesla
        System.out.println(car2.model); // Tesla
    }
}
```

(v) finalize() Method

The `finalize()` method is called before an object is garbage collected.

Example of `finalize()`

```
class Car8 {
    @Override
    protected void finalize() throws Throwable {
        System.out.println("Car object is destroyed");
    }
```

```
}


public class ObjectFinalizeExample {

   public static void main(String[] args) {

       Car8 car = new Car8();

       car = null;

       System.gc(); // Suggest garbage collection

   }

}
```

---

1. What is `instanceof` Operator in Java?

The `instanceof` operator is a **binary operator** in Java used to **test whether an object is an instance of a specific class or a subclass of that class**. It is used to check **type compatibility** at runtime.

Syntax:

```
object instanceof ClassName
```

- **Returns `true`** → If the object is an instance of the specified class or its subclass.
- **Returns `false`** → If the object is not an instance of the specified class.

Example of `instanceof` Operator:

```
class Animal1 {}
class Dog extends Animal1 {}

public class InstanceofExample {
   public static void main(String[] args) {
```

```
        Dog dog = new Dog();
        System.out.println(dog instanceof Dog);     // true
        System.out.println(dog instanceof Animal1); // true
        System.out.println(dog instanceof Object); // true
    }
}
```

✅ `instanceof` confirms that **dog** is an instance of **Dog**, **Animal**, and **Object**.

Why Use `instanceof` Operator?

- **Avoid ClassCastException** → Prevents invalid type casting.
- **Supports Runtime Type Checking** → Useful when dealing with polymorphism.
- **Ensures Safe Downcasting** → Checks before performing a downcast.