UNIT 4: Inheritance, Packages and Interfaces in Java

---

1. Inheritance in Java

1.1 Meaning of Inheritance

Inheritance is one of the most important features of **Object-Oriented Programming (OOP)**. It allows a **new class** to acquire the **properties and behaviors** of an existing class.

Exam Definition:

> **Inheritance is a mechanism in Java by which one class inherits the variables and methods of another class.**

Advantages of Inheritance

- **Code reusability**
- **Reduced redundancy**
- **Better organization**
- **Supports hierarchical classification**
- **Improves maintainability**

1.2 Using the `extends` Keyword

In Java, inheritance is implemented using the `extends` **keyword**.

Syntax

```
class SubClass extends SuperClass {
  // members
}
```

Example

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

class Dog extends Animal {
```

```
  void bark() {
    System.out.println("Dog barks");
  }
}
```

Here:

- `Animal` → **Superclass**
- `Dog` → **Subclass**

---

1. What is inheritance?
2. Which keyword is used to implement inheritance in Java?

Subclasses and Superclasses

Superclass

- The class whose properties are inherited
- Also called **parent class** or **base class**

Subclass

- The class that inherits properties
- Also called **child class** or **derived class**

Important Rule: A subclass can access **public** and **protected** members of the superclass.

Types of Inheritance Supported by Java

- Single inheritance ☑
- Multilevel inheritance ☑
- Hierarchical inheritance ☑ ❌ Multiple inheritance using classes is **not supported** (to avoid ambiguity).

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.

```
//Super class
class Vehicle {
  Vehicle() {
    System.out.println("This is a Vehicle");
  }
}

// Subclass
```

```
class Car extends Vehicle {
  Car() {
    System.out.println("This Vehicle is Car");
  }
}

public class Test {
  public static void main(String[] args) {
      // Creating object of subclass invokes base class constructor
    Car obj = new Car();
  }
}
```

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.

```
class Vehicle {
  Vehicle() {
    System.out.println("This is a Vehicle");
  }
}
class FourWheeler extends Vehicle {
  FourWheeler() {
    System.out.println("4 Wheeler Vehicles");
  }
}
class Car extends FourWheeler {
  Car() {
    System.out.println("This 4 Wheeler Vehicle is a Car");
  }
}
public class Geeks {
  public static void main(String[] args) {
    Car obj = new Car(); // Triggers all constructors in order
  }
}
```

3. Hierarchical Inheritance

In hierarchical inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class. For example, cars and buses both are vehicle

```
class Vehicle {
  Vehicle() {
    System.out.println("This is a Vehicle");
  }
```

```
}

class Car extends Vehicle {
  Car() {
    System.out.println("This Vehicle is Car");
  }
}

class Bus extends Vehicle {
  Bus() {
    System.out.println("This Vehicle is Bus");
  }
}

public class Test {
  public static void main(String[] args) {
    Car obj1 = new Car();
    Bus obj2 = new Bus();
  }
}
```

4. Hybrid Inheritance

It is a mix of two or more of the above types of inheritance. In Java, we can achieve hybrid inheritance only through Interfaces if we want to involve multiple inheritance to implement Hybrid inheritance.

- class Car extends Vehicle->Single Inheritance
- class Bus extends Vehicle and class Bus implements Interface Fare->Hybrid Inheritance (since Bus inherits from two sources, forming a combination of single + multiple inheritance).

The `super` Keyword

The `super` **keyword** is used to refer to the **immediate parent class object**.

Uses of `super` Keyword

1. Access superclass variables

```
super.variableName;
```

2. Call superclass methods

```
super.methodName();
```

3. Call superclass constructor

```
super();
```

Example

```
class Univeristy{
  String universityName = "TU";

  public University() {
    System.out.println("I am University");
  }

  public void displayUniversityName() {
    System.out.println("The univerisy name is " + universityName);
  }
}

class College extends University {
  String collegeName = "Golden Gate";

  public College() {
    super(); //Call superclass constructor
    super.displayUniversityName(); //Call superclass methods
    System.out.println(super.universityName); //Access superclass variables
    System.out.prinln("I am College");
  }
}
```

Method Overriding

Meaning of Method Overriding

Method overriding occurs when:

- A subclass provides a **specific implementation** of a method already defined in its superclass.

> Method overriding allows a subclass to redefine a method of its superclass.

Rules for Method Overriding

- Method name must be the same
- Parameter list must be the same
- Inheritance must exist
- Return type must be same
- Access level cannot be reduced

Example

```
class Animal {
  void sound() {
    System.out.println("Animal makes sound");
  }
}

class Dog extends Animal {
  void sound() {
    System.out.println("Dog barks");
  }
}
```

---

1. What is method overriding?
2. Can static methods be overridden?

Dynamic Method Dispatch

Meaning

Dynamic Method Dispatch is a mechanism by which a **call to an overridden method** is resolved **at runtime** rather than compile time.

This supports **runtime polymorphism**.

Example

```
class Animal {
  void sound() {
    System.out.println("Animal makes sound");
  }
}

class Dog extends Animal {
  void sound() {
    System.out.println("Dog barks");
  }
}
public class TestDispatch {
  public static void main(String[] args) {
    Animal a;
    a = new Dog();
    a.sound();
  }
}
```

Output: Dog barks

The method is decided based on **object type**, not reference type. Since the object type is of Dog() not Animal(), hence sound() of Dog is called.

---

1. What is dynamic method dispatch?
2. Is it resolved at compile time or runtime?

---

2. The Object Class

Meaning

The **Object class** is the **root class** of all Java classes.

Every class in Java implicitly extends `Object`.

Common Methods of Object Class

- `toString()`
- `equals()`
- `hashCode()`
- `getClass()`
- `clone()`

Example

```
class Test {
}
```

Internally:

```
class Test extends Object {
}
```

3. Abstract Classes

Meaning

An **abstract class** is a class that:

- Cannot be instantiated
- May contain **abstract methods**

Declared using the `abstract` keyword.

Abstract Method

A method without a body.

```
abstract void show();
```

Example

```
abstract class Shape {
  abstract void draw();
}

class Circle extends Shape {
  void draw() {
      System.out.println("Drawing Circle");
  }
}
```

Rules

- Abstract class may contain concrete methods
- Subclass must implement all abstract methods
- Constructors are allowed

---

1. Can we create an object of abstract class?
2. Why are abstract classes used?

---

4. Final Classes

Final Class

A class declared with `final` cannot be inherited.

```
final class Security {
}
```

Used for **security and performance**

Final Method

Cannot be overridden.

Final Variable

Value cannot be changed (constant).

---

1. Why is a class declared final?
2. Can final methods be overridden?

---

5. Packages in Java

Meaning of Package

A **package** is a **group of related classes and interfaces**.

Used for:

- Code organization
- Reusability
- Access protection

5.1 Defining a Package

Syntax

```
package mypackage;
```

5.2 Importing a Package

Syntax

```
import packageName.ClassName;
import packageName.*;
```

Example

[src/com/mycompany/mathutils/Calculator.java]

```
package com.mycompany.mathutils;

public class Calculator {
```

```
  public static int add(int a, int b) {
    return a + b;
  }
  public static int subtract(int a, int b) {
    return a - b;
  }

}
```

[TestPackage.java]

```
import com.mycompany.mathutils.Calculator;

public class TestPackage {
  public static void main(String[] args) {
    int sum = Calculator.add(10, 5);
    int difference = Calculator.subtract(10, 5);

    System.out.println("10 + 5 = " + sum);
    System.out.println("10 - 5 = " + difference);
  }
}
```

5.3 Access Control

Access modifiers determine **visibility** of class members (variables, methods, constructors, inner classes) and control **who can access them**.

There are **four types**:

1. **private**
2. **default** (no modifier)
3. **protected**
4. **public**

```
package mypack1;

public class P1 {
  private int a = 10;
  int b = 20; // default
  protected int c = 30;
  public int d = 40;
}

package mypack1;

public class P2 extends P1 {
  public void show() {
```

```
    System.out.println(b); // default, accessible
    System.out.println(c); // protected, accessible
    System.out.println(d); // public, accessible
    // System.out.println(a); // private, NOT accessible
  }
}
```

- Subclass in the **same package** can access **default, protected, public**, but **not private**.

```
package mypack2;

import mypack1.P1;

public class P3 extends P1 {
  public void show() {
    // System.out.println(a); //  private, not accessible
    // System.out.println(b); //  default (different package) not accessible
    System.out.println(c);    //  protected (through inheritance) accessible
    System.out.println(d);    //  public acceessible
  }
}

public class P4 {
  public void show() {
    P1 obj = new P1();

    // System.out.println(obj.a); //  private not accessible
    // System.out.println(obj.b); //  default not accessible
    // System.out.println(obj.c); //  protected (not a subclass) not accessible
    System.out.println(obj.d);    //  public accessible
  }
}
```

1. What is the use of packages?
2. Which access modifier is most restrictive?

---

6. Interfaces in Java

- An **interface** is a special type of abstract class in Java where **all methods are abstract by default**.
- Interfaces are used **purely for polymorphism**—they do not provide implementation.

Declared using `interface` keyword.

Syntax

```
interface MyInterface {
    void show();
```

```
}
```

Rules

- Methods are public and abstract by default
- Variables are public, static, and final
- No constructors
- Interfaces cannot have concrete method implementations (except with default or static methods in modern Java)

Implementing Interfaces

A class implements an interface using `implements` keyword.

Example

```
interface Test1 {
  void method1(); // public & abstract by default
  void method2();
}

class Test2 implements Test1 {
  @Override
  public void method1() { System.out.println("Method 1 implemented"); }

  @Override
  public void method2() { System.out.println("Method 2 implemented"); }
}
```

Abstract class equivalent:

```
abstract class Test1 {
  abstract void method1();
  abstract void method2();
}

class Test2 extends Test1 {
  @Override
  void method1() { System.out.println("Method 1 implemented"); }

  @Override
  void method2() { System.out.println("Method 2 implemented"); }
}
```

Multiple Inheritance Using Interfaces

Java allows **multiple inheritance using interfaces**.

```
interface A { void methodA(); }
interface B { void methodB(); }

class C implements A, B {
  public void methodA() { System.out.println("A"); }
  public void methodB() { System.out.println("B"); }
}
```

---

1. Difference between interface and abstract class?
2. Can a class implement multiple interfaces?
