

UNDERGRADUATE LEARNING SERIES / UNIT II

JAVA OBJECT-ORIENTED PROGRAMMING

Inheritance, Interfaces एवं Packages

सरल Hindi-English व्याख्या के साथ

12 अध्यायों में Unit II की व्यवस्थित शिक्षा

INHERITANCE

INTERFACES

MCQs

Hindi + English Edition

Inheritance · Interfaces · Packages

पुस्तक का परिचय

How to Use This Book

यह पुस्तक आपके Unit II syllabus के आधार पर inheritance, superclass/subclass, polymorphism, method overloading/overriding, abstract classes, final, interfaces, interface extension, nested interfaces, packages, classpath और access protection को एक क्रम में प्रस्तुत करती है।

व्याख्या Hindi-English mix में है। Technical terms, Java syntax और commands English में रखी गई हैं। प्रत्येक अध्याय में explanations, examples, outputs, revision और अभ्यास दिए गए हैं।

पढ़ने का तरीका

पहले concept पढ़ें, फिर program स्वयं type करके चलाएँ। Output बदलने के लिए input values बदलें। MCQs हल करने के बाद answer key देखें। केवल code याद करने के स्थान पर execution की प्रक्रिया समझें।

Version और Program Conventions

अधिकांश examples traditional class-based Java syntax पर आधारित हैं। Examples Java 17-compatible traditional forms पर आधारित हैं। Java 8/9 की interface सुविधाओं का जहाँ उल्लेख है, वहीं version बताया गया है। Newer version-specific syntax इस unit के basic examples में उपयोग नहीं की गई है।

Public class वाला program उसी class name की .java file में save करें। छोटे fragments किसी उचित method/class के भीतर उपयोग करें। अलग examples में समान helper class names हो सकते हैं; उन्हें अलग folders में रखें।

Symbols और Design

नीले boxes में code और commands, हल्के हरे tables में comparisons और पीले boxes में exam points दिए गए हैं। अध्याय 12 पूरे unit का revision, MCQ answer key और mini project प्रदान करता है।

विषय-सूची

Contents / Chapters 1-12

01. Inheritance का परिचय	4
02. Types of Inheritance	8
03. Multilevel Inheritance में Constructors	12
04. Polymorphism एवं Method Overloading	16
05. Method Overriding	19
06. Abstract Classes	23
07. Inheritance में final	27
08. Interfaces	30
09. Extending एवं Nested Interfaces	34
10. Packages का परिचय एवं निर्माण	38
11. Classpath, Imports एवं Access Protection	41
12. Revision एवं Question Bank	45

UNIT II / CHAPTER 01

अध्याय 1: Inheritance का परिचय

Superclass, Subclass एवं extends

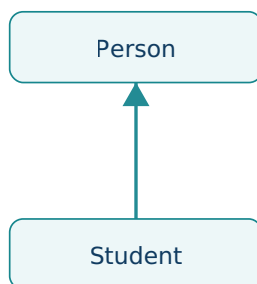


Figure 1.1: Student extends Person; arrow points toward superclass.

1.1 Learning Objectives

इस अध्याय में inheritance का अर्थ, superclass और subclass का संबंध, inherited members, extends syntax तथा is-a relationship समझेंगे। Inheritance और composition के चुनाव का भी परिचय मिलेगा।

1.2 Inheritance क्या है?

Inheritance वह mechanism है जिसके द्वारा एक नई class किसी existing class से संबंध स्थापित करके उसके inheritable members का उपयोग करती है। Existing class superclass, base class या parent class कहलाती है। नई class subclass, derived class या child class कहलाती है।

उदाहरण: Person में name और displayName() हों। Student, Person का एक विशेष प्रकार है और उसमें rollNumber भी हो सकता है। Student हर Person नहीं है; बल्कि हर Student एक Person माना जा सकता है। यह **is-a relationship** है।

शब्द	अर्थ	उदाहरण
Superclass	General type	Person
Subclass	More specific type	Student
Inherited member	नियमों के अनुसार प्राप्त member	displayName()
Extension	नई functionality	rollNumber

1.3 Syntax

```
class Subclass extends Superclass {  
    // Additional fields and methods  
}
```

एक class की केवल एक direct superclass होती है। यदि explicit extends नहीं है, तो सामान्य class का superclass Object होता है; Object स्वयं अपवाद है।

1.4 Complete Program

Save as InheritanceDemo.java:

```
class Person {  
    String name;  
  
    void displayName() {  
        System.out.println("Name: " + name);  
    }  
}  
  
class Student extends Person {  
    int rollNumber;  
  
    void displayStudent() {  
        displayName();  
        System.out.println("Roll number: " + rollNumber);  
    }  
}  
  
public class InheritanceDemo {  
    public static void main(String[] args) {  
        Student student = new Student();  
        student.name = "Amit";  
        student.rollNumber = 101;  
        student.displayStudent();  
    }  
}
```

Output:

```
Name: Amit  
Roll number: 101
```

Student में name और displayName() दोबारा declare नहीं किए गए। इस same-package example में वे inherited और accessible हैं।

1.5 Inherited Members के नियम

- Constructors inherited नहीं होते।
- Superclass के private members subclass द्वारा inherited नहीं होते।

- Private state superclass के accessible methods से उपयोग की जा सकती है।
- Public और protected members inheritance में महत्वपूर्ण हैं।
- Package-private access package boundary पर निर्भर है।
- Final class को extend नहीं कर सकते।

महत्वपूर्ण: “Subclass superclass की हर चीज सीधे access करती है” गलत है। Inheritance, accessibility और object की internal state अलग अवधारणाएँ हैं।

1.6 Private Field का सुरक्षित उपयोग

```
class Person {
    private String name = "Amit";

    public String getName() {
        return name;
    }
}
class Student extends Person {
    void display() {
        System.out.println(getName());
    }
}
```

Student सीधे name field नहीं पढ़ सकती, लेकिन public getter उपयोग कर सकती है। Encapsulation टूटे बिना behavior reuse होता है।

1.7 Superclass Reference और Subclass Object

```
Person person = new Student();
```

यह upcasting का उदाहरण है। Reference का declared type Person है; actual object Student है। Accessible operations compile time पर reference type के अनुसार जाँची जाती हैं। Overridden instance methods का चयन runtime object के अनुसार हो सकता है।

Person reference से Student-only method सीधे call नहीं होगी। यह विषय runtime polymorphism chapter में विस्तार से पढ़ेंगे।

1.8 Inheritance बनाम Composition

आधार	Inheritance	Composition
संबंध	is-a	has-a
उदाहरण	Student is a Person	College has a Library
Implementation	extends	Field/object reference

आधार	Inheritance	Composition
उपयुक्तता	वास्तविक subtype संबंध	Object collaboration

केवल कुछ code reuse करने के लिए गलत is-a संबंध न बनाएं। Car has an Engine है; Car को Engine extend करना उचित model नहीं है।

1.9 Quick Revision एवं अभ्यास

Inheritance reuse और specialization में सहायता करती है। Extends class hierarchy बनाता है। Constructors inherit नहीं होते। Private fields के लिए accessible methods उपयोग करें।

लघु प्रश्न: Superclass और subclass क्या हैं? Is-a relationship उदाहरण सहित लिखिए।

दीर्घ प्रश्न: Inheritance के benefits एवं limitations समझाइए। Inheritance और composition की तुलना कीजिए।

Programming: Employee superclass तथा Teacher subclass बनाइए। Name, salary और subject प्रदर्शित कीजिए।

MCQs: 1. Class inheritance keyword: A. implements B. extends C. import D. final 2. Constructors: A. हमेशा inherited B. inherited नहीं C. केवल private inherited D. fields हैं 3. Student-Person संबंध: A. has-a B. is-a C. part-of only D. कोई नहीं

उत्तर: 1-B, 2-B, 3-B.

UNIT II / CHAPTER 02

अध्याय 2: Types of Inheritance

Single, Multilevel, Hierarchical एवं Interface-Based Multiple Inheritance

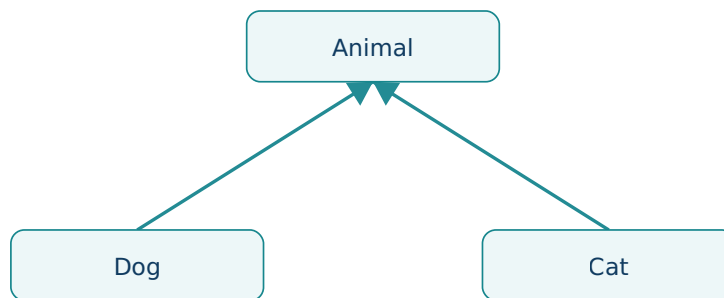


Figure 2.1: Hierarchical inheritance; Dog and Cat share Animal.

2.1 Learning Objectives

विभिन्न inheritance structures पहचानेंगे, Java class inheritance की सीमाएँ समझेंगे और interfaces के माध्यम से multiple type inheritance का अर्थ जानेंगे।

2.2 Single Inheritance

एक subclass एक superclass से derive होती है।

```
class Animal {  
    void eat() {  
        System.out.println("Eating");  
    }  
}  
class Dog extends Animal {  
    void bark() {  
        System.out.println("Barking");  
    }  
}
```

Dog object inherited eat() और अपनी bark() call कर सकता है।

2.3 Multilevel Inheritance

Inheritance chain में एक derived class आगे superclass बनती है।

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

```

    }
}
class Mammal extends Animal {
    void walk() {
        System.out.println("Walking");
    }
}
class Dog extends Mammal {
    void bark() {
        System.out.println("Barking");
    }
}

```

Dog, Mammal और Animal दोनों का subtype है। Dog object eat(), walk() और bark() उपयोग कर सकता है, यदि access rules अनुमति दें।

2.4 Hierarchical Inheritance

एक superclass से अनेक subclasses बनती हैं।

```

class Shape {
    void describe() {
        System.out.println("A geometric shape");
    }
}
class Circle extends Shape {
}
class Rectangle extends Shape {
}

```

Circle और Rectangle sibling classes हैं। किसी एक का दूसरे से automatic subtype संबंध नहीं बनता।

2.5 Multiple Class Inheritance की सीमा

Java में यह invalid है:

```

// INVALID: two direct superclass names
// class C extends A, B { }

```

एक class दो classes को directly extend नहीं कर सकती। यह ambiguity और implementation complexity को सीमित करता है।

फिर भी class अनेक interfaces implement कर सकती है। इसलिए “Java में किसी प्रकार की multiple inheritance नहीं है” भी अधूरा कथन है।

2.6 Interfaces द्वारा Multiple Type Inheritance

```

interface Printable {
    void print();
}
interface Scannable {
    void scan();
}

```

```

}
class MultiFunctionDevice implements Printable, Scannable {
    public void print() {
        System.out.println("Printing");
    }
    public void scan() {
        System.out.println("Scanning");
    }
}
public class MultipleTypeDemo {
    public static void main(String[] args) {
        MultiFunctionDevice device = new MultiFunctionDevice();
        device.print();
        device.scan();
    }
}

```

Output:

```

Printing
Scanning

```

एक object दोनों interface contracts पूरा करता है। Default methods की conflicts को explicitly resolve करना पड़ सकता है; interfaces chapters में पढ़ेंगे।

2.7 Hybrid Structure

Hybrid शब्द अनेक structures के combination के लिए उपयोग होता है। Java classes में single superclass rule फिर भी लागू रहता है। Classes और interfaces मिलकर complex type hierarchy बना सकते हैं।

```

class Machine { }
interface Printable { void print(); }
interface Scannable { void scan(); }
class Device extends Machine implements Printable, Scannable {
    public void print() { }
    public void scan() { }
}

```

यह class extension और multiple interface implementation का combination है।

2.8 Comparison

Structure	अर्थ	Java में support
Single	One parent, one child relation	Classes
Multilevel	Chain of inheritance	Classes
Hierarchical	One parent, many children	Classes
Multiple direct class	One child, many class parents	नहीं

Structure	अर्थ	Java में support
Multiple interface types	Many interface contracts	हाँ

2.9 Object Class

सामान्य Java class hierarchy में Object root है। ToString(), equals() और hashCode() जैसी methods Object से संबंधित हैं। सभी primitive values Object instances नहीं हैं। Arrays objects हैं, लेकिन primitive int स्वयं object नहीं है।

2.10 अभ्यास एवं MCQs

अभ्यास: Animal-Mammal-Dog chain बनाइए। Vehicle से Car और Bike derive कीजिए। Printer और Scanner contracts implement कीजिए। प्रत्येक structure का diagram बनाइए।

1. One parent से many subclasses: A. Multilevel B. Hierarchical C. Single only D. Multiple class
2. Java class की direct superclass संख्या: A. 0 या 1 B. हमेशा 2 C. Unlimited D. 3
3. Class many interfaces: A. नहीं B. implement कर सकती है C. केवल private D. केवल final होने पर

उत्तर: 1-B, 2-A, 3-B.

UNIT II / CHAPTER 03

अध्याय 3: Multilevel Inheritance में Constructors

super(), this() एवं Initialization Order

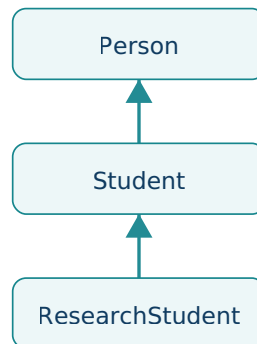


Figure 3.1: Multilevel hierarchy; constructor bodies initialize from top to bottom.

3.1 Learning Objectives

Constructor chain, superclass initialization, explicit super arguments और initialization order समझेंगे। Constructor inherited होने और invoke होने का अंतर स्पष्ट करेंगे।

3.2 Constructors Inherited नहीं होते

Subclass object बनाने पर superclass constructor invoke हो सकता है, लेकिन उसका constructor subclass का inherited member नहीं बनता। प्रत्येक class अपनी initialization responsibility रखती है।

3.3 Basic Execution Order

```
class A {
    A() {
        System.out.println("A constructor");
    }
}
class B extends A {
    B() {
        System.out.println("B constructor");
    }
}
class C extends B {
    C() {
        System.out.println("C constructor");
    }
}
public class ConstructorOrderDemo {
    public static void main(String[] args) {
```

```

        new C();
    }
}

```

Output:

```

A constructor
B constructor
C constructor

```

C creation में superclass initialization पहले होती है। तीन constructor calls का अर्थ तीन अलग A/B/C objects बनना नहीं है; यह एक C object की class hierarchy का initialization है।

3.4 super() और Implicit Invocation

यदि explicit this(...) या super(...) invocation नहीं लिखी गई, तो ordinary constructors में implicit super() शामिल होता है। Accessible no-argument superclass constructor चाहिए।

```

class Parent {
    Parent(int value) { }
}
// INVALID unless a suitable super(...) call is added:
// class Child extends Parent { Child() { } }

```

Parent में केवल parameterized constructor होने पर automatic no-argument constructor नहीं मिलता।

3.5 Parameterized Multilevel Program

```

class Person {
    private final String name;
    Person(String name) {
        this.name = name;
        System.out.println("Person initialized");
    }
    String getName() { return name; }
}
class Student extends Person {
    private final int roll;
    Student(String name, int roll) {
        super(name);
        this.roll = roll;
        System.out.println("Student initialized");
    }
    int getRoll() { return roll; }
}
class ResearchStudent extends Student {
    private final String topic;
    ResearchStudent(String name, int roll, String topic) {
        super(name, roll);
        this.topic = topic;
        System.out.println("ResearchStudent initialized");
    }
    void display() {

```

```

        System.out.println(getName() + ", " + getRoll());
        System.out.println("Topic: " + topic);
    }
}
public class MultilevelConstructorDemo {
    public static void main(String[] args) {
        ResearchStudent student =
            new ResearchStudent("Amit", 101, "Java");
        student.display();
    }
}

```

Output:

```

Person initialized
Student initialized
ResearchStudent initialized
Amit, 101
Topic: Java

```

3.6 this(...) बनाम super(...)

Invocation	Target
this(...)	Same-class constructor
super(...)	Direct superclass constructor
super.method()	Superclass method implementation

इस पुस्तक के examples explicit constructor invocation पहले statement में रखते हैं। यह Java 17-compatible traditional pattern है। एक constructor में this(...) और super(...) दोनों explicit chaining invocations साथ नहीं लिखते। This chain अंततः superclass construction तक पहुँचती है। Recursive constructor invocation invalid है।

3.7 Field Initialization Order

Ordinary construction में object fields पहले default values पाते हैं। Constructor chain superclass की initialization करती है; subclass instance field initializers और initializer blocks declaration order में चलते हैं, फिर subclass constructor body का शेष भाग चलता है।

```

class Parent {
    Parent() { System.out.println("Parent body"); }
}
class Child extends Parent {
    int number = show();
    int show() {
        System.out.println("Child field initializer");
        return 10;
    }
}

```

```
Child() { System.out.println("Child body"); }  
}
```

Child creation पर क्रम: Parent body, Child field initializer, Child body।

3.8 Constructor में Overridable Method का जोखिम

Superclass constructor यदि overridden method call करे, तो subclass implementation subclass initialization पूरी होने से पहले चल सकती है। इसलिए constructors में overridable behavior call करने से बचना एक उपयोगी design practice है।

3.9 Revision एवं अभ्यास

Super केवल direct superclass constructor invoke करता है; उस superclass की chain आगे जाती है। Constructors inherited नहीं होते। Parent no-argument constructor unavailable हो तो suitable super(args) लिखें।

अभ्यास: Person-Employee-Teacher chain में parameterized constructors बनाएँ। Constructor outputs से order दिखाएँ।

MCQs: 1. C extends B, B extends A में bodies का order: A. C-B-A B. A-B-C C. B-C-A D. Random 2. super(args) invokes: A. Direct superclass constructor B. Package C. Static method only D. JVM 3. Constructor inherited है? A. हाँ B. नहीं C. केवल public D. केवल abstract

उत्तर: 1-B, 2-A, 3-B.

UNIT II / CHAPTER 04

अध्याय 4: Polymorphism एवं Method Overloading

Compile-Time Selection of Methods

4.1 Polymorphism

Polymorphism का अर्थ एक सामान्य operation या type के माध्यम से अलग forms का उपयोग करना है।

Java teaching में overloading को compile-time polymorphism और overriding को runtime polymorphism कहा जाता है।

Overloading में method name समान रहता है, लेकिन parameter list अलग होती है। Compiler argument types और applicable rules से method चुनता है।

4.2 Overloading के नियम

परिवर्तन	Overloading संभव?
Parameters की संख्या	हाँ
Parameter types	हाँ
Types का क्रम	हाँ, यदि signature अलग हो
केवल parameter names	नहीं
केवल return type	नहीं

4.3 Complete Program

```
public class OverloadingDemo {
    static int add(int a, int b) {
        return a + b;
    }
    static double add(double a, double b) {
        return a + b;
    }
    static int add(int a, int b, int c) {
        return a + b + c;
    }
    public static void main(String[] args) {
        System.out.println(add(10, 20));
        System.out.println(add(2.5, 3.5));
        System.out.println(add(10, 20, 30));
    }
}
```

Output:

```
30
6.0
60
```

4.4 Return Type Alone पर्याप्त नहीं

```
// INVALID as a pair:
// int calculate(int value) { return value; }
// double calculate(int value) { return value; }
```

Same name और same parameter types होने पर return type से अलग overload नहीं बनता।

4.5 Widening और Applicable Methods

```
static void show(long value) {
    System.out.println("long");
}
static void show(Integer value) {
    System.out.println("Integer");
}
```

show(10) के लिए output long है। Applicable primitive widening method boxing phase से पहले consider होती है। Overload resolution केवल “जिस type का size छोटा हो वही चुना जाए” जैसा नियम नहीं है।

4.6 Autoboxing और Varargs

Autoboxing primitive को wrapper representation में convert कर सकती है। Varargs variable संख्या के arguments लेती है।

```
static int sum(int... values) {
    int total = 0;
    for (int value : values) { total += value; }
    return total;
}
```

sum(10,20,30) returns 60। Method के भीतर values int array है। Varargs parameter last position में होना चाहिए। Fixed-arity applicable methods सामान्यतः varargs invocation phase से पहले चुनी जाती हैं।

4.7 Ambiguous Call

```
static void show(String value) { }
static void show(Integer value) { }
// show(null); // Compilation error: ambiguous
```

String और Integer unrelated reference types हैं; null दोनों के लिए applicable है। Explicit cast जैसे `show((String)null)` target स्पष्ट कर सकता है।

4.8 Declared Type का महत्व

```
static void show(Object value) {  
    System.out.println("Object");  
}  
static void show(String value) {  
    System.out.println("String");  
}
```

`Object value="Java"; show(value);` में Object overload चुनी जाती है। Actual String object होने से overload runtime पर String overload में नहीं बदलती।

4.9 Overloading के लाभ और सीमाएँ

Related operations के लिए readable API मिलती है। लेकिन unrelated argument types, null और conversions वाली बहुत overloads confusion पैदा कर सकती हैं। Parameters का meaningful design रखें।

4.10 Revision और अभ्यास

Overloading inheritance के बिना भी संभव है। Static और instance methods दोनों overload हो सकती हैं। Constructors भी overload हो सकते हैं। Return type alone पर्याप्त नहीं है।

अभ्यास: Area के overloads बनाएँ: square side, rectangle length/width और circle radius double। अलग types/count रखें।

MCQs: 1. Overloading selection: A. Compile time B. केवल runtime C. GC D. Package loading 2. केवल return type बदलना: A. पर्याप्त B. पर्याप्त नहीं C. हमेशा static D. Constructor 3. String/Integer overload में `show(null)`: A. String हमेशा B. Integer हमेशा C. Ambiguous D. दोनों चलेंगी

उत्तर: 1-A, 2-B, 3-C.

UNIT II / CHAPTER 05

अध्याय 5: Method Overriding

Runtime Polymorphism एवं Dynamic Method Dispatch

5.1 Learning Objectives

Subclass द्वारा inherited behavior को specialize करना सीखेंगे। Reference type, runtime object, super calls, covariant returns और overriding restrictions समझेंगे।

5.2 Overriding क्या है?

Subclass superclass की overridable instance method के लिए compatible declaration देती है, तो overriding होती है। Same method name और compatible signature होती है; method का implementation बदल सकता है।

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

@Override annotation compiler को intended overriding जाँचने देती है। Annotation हटाने से valid overriding बंद नहीं होती, लेकिन annotation mistakes पकड़ने में सहायता करती है।

5.3 Complete Runtime Polymorphism Program

```
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Bark");
    }
}
class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Meow");
    }
}
public class DispatchDemo {
```

```

    public static void main(String[] args) {
        Animal animal = new Dog();
        animal.sound();
        animal = new Cat();
        animal.sound();
    }
}

```

Output:

```

Bark
Meow

```

Animal reference समान है, actual object अलग है। Overridden instance method का implementation runtime object से चुना जाता है।

5.4 Compile-Time और Runtime Responsibilities

Compiler reference type में method की availability, arguments और access जाँचता है। Runtime selected instance method का overriding implementation चुन सकता है। Subclass-only method superclass reference से सीधे call नहीं होती।

5.5 Overriding Rules

विषय	नियम
Signature	Override-compatible signature
Return type	Same या allowed covariant reference return
Access	अधिक restrictive नहीं कर सकते
final method	Override नहीं
private method	Inherited नहीं; override नहीं
static method	Hide होती है; instance override नहीं
Checked exceptions	नया broader checked exception नहीं

Public method को protected/package-private/private बनाना invalid है। Primitive return type covariant नहीं होती; int return को long में बदलकर override नहीं कर सकते।

5.6 Covariant Return Type

```

class Parent {
    Parent copy() { return new Parent(); }
}
class Child extends Parent {

```

```
@Override
Child copy() { return new Child(); }
}
```

Child return type Parent का subtype है, इसलिए allowed है। यह return type change alone वाला method overload नहीं है; यह inherited instance method की overriding है।

5.7 super से Parent Implementation

```
class Parent {
    void display() {
        System.out.println("Parent");
    }
}
class Child extends Parent {
    @Override
    void display() {
        super.display();
        System.out.println("Child");
    }
}
```

Child display call का output Parent, फिर Child है। Super call parent behavior के साथ extension करने में उपयोगी है।

5.8 Static Methods और Field Hiding

```
class Parent {
    int value = 10;
    static void show() { System.out.println("Parent static"); }
}
class Child extends Parent {
    int value = 20;
    static void show() { System.out.println("Child static"); }
}
```

Parent p=new Child(); पर p.value 10 है। Field access declared reference type से resolve होता है। Parent.show() Parent static implementation call करती है। Static methods को object reference से call करना legal हो सकता है, लेकिन class name से call अधिक स्पष्ट है।

5.9 Upcasting और Downcasting

```
Animal animal = new Dog(); // Upcast
Dog dog = (Dog) animal;    // Valid for this actual object
```

Incompatible downcast ClassCastException दे सकती है। Traditional safe check:

```
if (animal instanceof Dog) {
    Dog dog = (Dog) animal;
}
```

```
dog.sound();
}
```

Null का reference cast हो सकता है, लेकिन उस पर instance method call NullPointerException देगा।

5.10 Overloading बनाम Overriding

आधार	Overloading	Overriding
Parameters	अलग	Compatible signature
Inheritance	आवश्यक नहीं	Subtype relationship
Selection	Compile-time overload resolution	Runtime instance dispatch
उद्देश्य	Related signatures	Specialized behavior

5.11 Revision एवं अभ्यास

Fields और static methods runtime instance overriding जैसा behavior नहीं देतीं। Final/private methods override नहीं होतीं। @Override compiler verification में सहायता करती है।

अभ्यास: Shape में area(), Circle और Rectangle में override करें। Superclass reference array से विभिन्न objects की methods call करें।

MCQs: 1. Animal a=new Dog(); overridden sound किसकी? A. Animal B. Dog C. दोनों हमेशा D. Random
2. Static method: A. Instance override B. Hide हो सकती है C. Constructor D. Abstract हमेशा
3. Public overridden method का private access: A. Allowed B. Invalid C. Required D. GC-dependent

उत्तर: 1-B, 2-B, 3-B.

UNIT II / CHAPTER 06

अध्याय 6: Abstract Classes

Abstract Methods एवं Concrete Subclasses

6.1 Abstract Class क्या है?

Abstract class ऐसी class है जिसका **direct instance नहीं बनाया जा सकता**। यह subclasses के लिए common state, implemented behavior और incomplete operations का आधार बन सकती है।

```
abstract class Shape {
    abstract double area();
    void describe() {
        System.out.println("A geometric shape");
    }
}
```

Abstract method की body नहीं होती; declaration semicolon से समाप्त होती है।

6.2 प्रमुख नियम

- Abstract class में abstract methods हो सकती हैं, लेकिन आवश्यक नहीं।
- Abstract method वाली ordinary class को abstract घोषित करना चाहिए।
- Abstract class का constructor हो सकता है।
- Concrete subclass को inherited abstract methods implement करनी चाहिए।
- Abstract subclass कुछ abstract methods unimplemented छोड़ सकती है।
- Abstract और final class modifiers incompatible हैं।

6.3 Complete Program

```
abstract class Shape {
    private final String name;
    Shape(String name) {
        this.name = name;
    }
    abstract double area();
    void display() {
        System.out.println(name + " area: " + area());
    }
}

class Rectangle extends Shape {
    private final double length;
    private final double width;
    Rectangle(double length, double width) {
        super("Rectangle");
        this.length = length;
        this.width = width;
    }
    @Override
```

```

    double area() {
        return length * width;
    }
}
class Circle extends Shape {
    private final double radius;
    Circle(double radius) {
        super("Circle");
        this.radius = radius;
    }
    @Override
    double area() {
        return Math.PI * radius * radius;
    }
}
public class AbstractDemo {
    public static void main(String[] args) {
        Shape shape = new Rectangle(10, 5);
        shape.display();
        Shape circle = new Circle(2);
        System.out.printf("Circle area: %.2f%n", circle.area());
    }
}

```

Output:

```

Rectangle area: 50.0
Circle area: 12.57

```

Formatted decimal separator locale पर निर्भर हो सकता है; shown output dot-decimal environment मानता है।

6.4 Abstract Reference

```

Shape shape = new Rectangle(10, 5); // Valid
// Shape shape = new Shape("Shape"); // Invalid

```

Abstract type का reference बनाना allowed है। Direct abstract-class instance creation allowed नहीं है।

6.5 Abstract Constructor का अर्थ

Constructor स्वयं abstract नहीं हो सकता। Abstract class का ordinary constructor subclass object की superclass state initialize करता है। इसलिए “abstract class का constructor कभी नहीं चलता” गलत है।

6.6 Invalid Abstract Method Modifiers

```

// Invalid combinations for ordinary abstract methods:
// private abstract void work();
// static abstract void work();
// final abstract void work();

```

Private method subclass contract की तरह override नहीं होती। Static method instance dispatch नहीं देती। Final method override रोकती है। इसलिए ये abstract instance-method requirement से conflict करते हैं।

6.7 Abstract Class बिना Abstract Method

```
abstract class Base {
    void display() {
        System.out.println("Common behavior");
    }
}
```

यह valid है। Designer direct instantiation रोकना चाहता हो सकता है, भले सभी declared methods implemented हों।

6.8 Template-Style Behavior

```
abstract class Report {
    final void printReport() {
        System.out.println("Report begins");
        printBody();
        System.out.println("Report ends");
    }
    abstract void printBody();
}
```

Final common operation structure निर्धारित करता है। Subclasses body का implementation देती हैं। Constructor के बजाय normal post-construction method में polymorphic operation रखना अधिक सुरक्षित है।

6.9 Abstract और Concrete Class

आधार	Abstract Class	Concrete Class
Direct object	नहीं	Suitable constructor होने पर संभव
Abstract methods	संभव	कोई unimplemented abstract requirement नहीं
Fields	संभव	संभव
Constructors	संभव	संभव
उपयोग	Base abstraction	Usable implementation

6.10 Revision और अभ्यास

Abstract class partially implemented base हो सकती है। Abstract reference concrete object को refer कर सकती है। Concrete subclass remaining abstract methods implement करती है। Constructors abstract नहीं हो सकते।

अभ्यास: Employee abstract class में salary() बनाएँ। PermanentEmployee और ContractEmployee implementations दें।

MCQs: 1. Abstract class direct object: A. Always B. नहीं C. केवल no-arg D. केवल final 2. Abstract class constructor: A. संभव B. असंभव C. हमेशा abstract D. inherited 3. Abstract method body: A. आवश्यक B. नहीं होती C. केवल empty braces D. हमेशा static

उत्तर: 1-B, 2-A, 3-B.

UNIT II / CHAPTER 07

अध्याय 7: Inheritance में final

Final Variables, Methods एवं Classes

7.1 Learning Objectives

Final के तीन अलग uses समझेंगे: variable reassignment रोकना, method overriding रोकना और class extension रोकना। Final reference तथा immutability का अंतर स्पष्ट करेंगे।

7.2 Final Variable

```
final int MAX_MARKS = 100;  
// MAX_MARKS = 200; // Compilation error
```

Variable को definite assignment rules के अनुसार एक बार value मिल सकती है। Final local declaration और initialization अलग हो सकती है।

```
final int limit;  
limit = 10;  
System.out.println(limit);
```

Final fields के initialization के लिए field initializer, initializer block या applicable constructors उपयोग किए जा सकते हैं। Blank final instance field प्रत्येक constructor path पर initialize होना चाहिए।

7.3 Constructor में Blank Final Field

```
class Student {  
    private final int rollNumber;  
    Student(int rollNumber) {  
        this.rollNumber = rollNumber;  
    }  
    int getRollNumber() {  
        return rollNumber;  
    }  
}
```

Object बनने के बाद rollNumber reassign नहीं होगा। हर object की अपनी rollNumber value हो सकती है।

7.4 Final Reference बनाम Immutable Object

```
final int[] marks = {70, 80};  
marks[0] = 95; // Valid  
// marks = new int[]{10}; // Invalid
```

Final reference को दूसरे array पर assign नहीं कर सकते। Existing array के elements बदल सकते हैं। Immutable object का content नहीं बदलता; final reference अकेले immutability नहीं देता।

7.5 Compile-Time Constants

```
static final int MAX_SIZE = 100;
final int parsed = Integer.parseInt("100");
```

MAX_SIZE constant expression से initialized primitive final है। Parsed final है लेकिन compile-time constant नहीं। Final String भी constant variable हो सकती है यदि initializer constant expression है।

7.6 Final Method

```
class Parent {
    final void display() {
        System.out.println("Fixed display behavior");
    }
}
class Child extends Parent {
    void extra() {
        System.out.println("Additional behavior");
    }
}
public class FinalMethodDemo {
    public static void main(String[] args) {
        Child child = new Child();
        child.display();
        child.extra();
    }
}
```

Output:

```
Fixed display behavior
Additional behavior
```

Final method inherited हो सकती है लेकिन override नहीं हो सकती। अलग parameter list वाली overload संभव हो सकती है। Final का अर्थ method call निषिद्ध नहीं है।

7.7 Final Class

```
final class Utility {
    static int square(int value) {
        return value * value;
    }
}
// class AdvancedUtility extends Utility { } // Invalid
```

Final class extend नहीं कर सकते। Final class का object suitable constructor के साथ बन सकता है। Final class interfaces implement कर सकती है।

7.8 Final Class का State

```
final class Counter {
    int value;
    void increment() { value++; }
}
```

यह mutable class है। Final class होने से हर object immutable नहीं हो जाता। Immutable design के लिए mutable state और exposure नियंत्रित करना पड़ता है।

7.9 Final, Finally और Finalize

शब्द	भूमिका
final	Modifier: reassignment/override/extension restrictions
finally	Exception-handling block
finalize()	Legacy finalization mechanism; deprecated, नए designs में उपयोग न करें

ये समान नाम वाले लेकिन अलग concepts हैं। Finally या finalization का final fields से direct संबंध नहीं है।

7.10 Common Mistakes एवं अभ्यास

Final method और abstract method incompatible हैं। Abstract final class impossible combination है। Final constructor घोषित नहीं कर सकते। Final references internal object mutation नहीं रोकतीं।

अभ्यास: Student identity का final field बनाएँ। Parent final method call करें। Mutable final reference का example दिखाएँ।

MCQs: 1. Final method: A. Override कर सकते हैं B. Override नहीं C. Call नहीं D. Constructor 2. Final class: A. Extend नहीं B. Object नहीं C. Fields नहीं D. Interface implement नहीं 3. Final array reference: A. Elements कभी नहीं बदलते B. Reference reassign नहीं C. Length बढ़ती D. Always null

उत्तर: 1-B, 2-A, 3-B.

UNIT II / CHAPTER 08

अध्याय 8: Interfaces

Definition, Implementation एवं Contracts

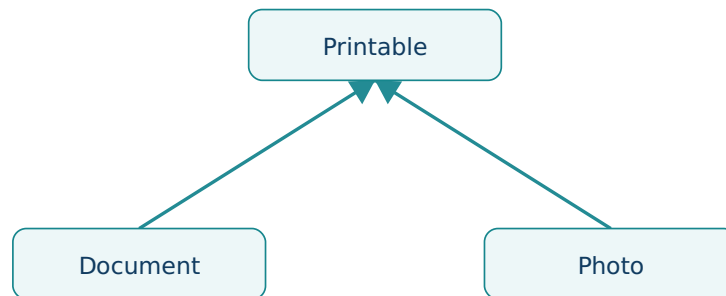


Figure 8.1: Two classes implement the Printable contract.

8.1 Interface क्या है?

Interface एक reference type है जो implementing classes के लिए operations का contract निर्धारित कर सकती है। अलग class hierarchies के objects समान interface के माध्यम से उपयोग हो सकते हैं।

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

इस declaration में print() implicitly public abstract है। Interface reference बन सकती है, लेकिन interface का direct instance नहीं बनता।

8.2 implements Keyword

```
class Document implements Printable {  
    @Override  
    public void print() {  
        System.out.println("Printing document");  
    }  
}
```

Implementing method public होनी चाहिए। Package-private implementation से public interface contract पूरा नहीं होता। Concrete class required abstract methods implement करती है; नहीं तो class को abstract होना पड़ सकता है।

8.3 Complete Program

```
interface Printable {
    void print();
}
class Document implements Printable {
    public void print() {
        System.out.println("Document printed");
    }
}
class Photo implements Printable {
    public void print() {
        System.out.println("Photo printed");
    }
}
public class InterfaceDemo {
    public static void main(String[] args) {
        Printable[] items = {new Document(), new Photo()};
        for (Printable item : items) {
            item.print();
        }
    }
}
```

Output:

```
Document printed
Photo printed
```

Caller को concrete implementation details जानना आवश्यक नहीं।

8.4 Interface Fields

```
interface ExamRules {
    int PASS_MARKS = 40;
}
```

Interface fields implicitly public static final हैं। Initializer चाहिए। लेकिन हर interface final field compile-time constant नहीं; initializer constant expression होना भी आवश्यक है। Mutable object का final reference interface field में रखना उसके object को immutable नहीं बनाता।

8.5 Methods के प्रकार

Method form	Version / व्यवहार
Ordinary abstract	Public abstract contract
default	Java 8 से instance implementation
static	Java 8 से interface-related helper

Method form	Version / व्यवहार
private	Java 9 से internal helper implementation

Interfaces में constructors और ordinary instance fields नहीं होते। यह कहना कि “interface की सभी methods हमेशा abstract हैं” modern Java के लिए गलत है।

8.6 Default और Static Methods

```
interface Greeting {
    void greet();
    default void welcome() {
        System.out.println("Welcome");
    }
    static String language() {
        return "English";
    }
}
class UserGreeting implements Greeting {
    public void greet() {
        System.out.println("Hello");
    }
}
```

Object welcome() उपयोग कर सकता है। Static method को `Greeting.language()` से call करें। Interface static methods implementing class की inherited instance methods नहीं हैं।

8.7 Abstract Class बनाम Interface

आधार	Abstract Class	Interface
Class extension	One direct superclass	Many interfaces implement कर सकते हैं
Instance fields	संभव	नहीं
Constructor	संभव	नहीं
Behavior	Concrete/abstract methods	Abstract/default/static/private forms
मुख्य use	Shared base implementation/state	Contract और type abstraction

Class दोनों उपयोग कर सकती है:

```
class Machine { }
interface Printable { void print(); }
class Printer extends Machine implements Printable {
    public void print() { }
}
```

8.8 Interface का महत्त्व

Loose coupling में सहायता: code concrete class के बजाय contract पर निर्भर होता है। अलग implementations, testing substitutes और multiple capabilities को व्यवस्थित करने में उपयोगी है। Interface contract का quality design उसके नाम से नहीं, operations की स्पष्टता से आता है।

8.9 Revision एवं अभ्यास

Implements class-interface संबंध बनाता है। Interface fields public static final हैं। Public abstract methods के implementation में public access रखें। Default methods overridable behavior दे सकती हैं।

अभ्यास: Payable interface में amount() method बनाएँ; EmployeePayment और Invoice implement करें।

MCQs: 1. Class-interface keyword: A. extends only B. implements C. import D. package 2. Interface fields: A. private instance B. public static final C. protected only D. volatile हमेशा 3. Interface constructor: A. Required B. उपलब्ध नहीं C. inherited D. हमेशा default

उत्तर: 1-B, 2-B, 3-B.

UNIT II / CHAPTER 09

अध्याय 9: Extending एवं Nested Interfaces

Interface Inheritance, Default Conflicts एवं Importance

9.1 Learning Objectives

Interfaces की inheritance, multiple superinterfaces, nested interface qualification और default-method conflict resolution समझेंगे।

9.2 Interface Extends Interface

```
interface Printable {  
    void print();  
}  
interface AdvancedPrintable extends Printable {  
    void preview();  
}
```

AdvancedPrintable contract में inherited print() और declared preview() दोनों हैं। Interface दूसरी interface को extends करती है, implements नहीं।

9.3 Multiple Interface Inheritance

```
interface Readable {  
    void read();  
}  
interface Writable {  
    void write();  
}  
interface ReadWritable extends Readable, Writable {  
    void save();  
}  
class FileDocument implements ReadWritable {  
    public void read() {  
        System.out.println("Reading");  
    }  
    public void write() {  
        System.out.println("Writing");  
    }  
    public void save() {  
        System.out.println("Saving");  
    }  
}  
public class ExtendedInterfaceDemo {  
    public static void main(String[] args) {  
        ReadWritable document = new FileDocument();  
        document.read();  
        document.write();  
        document.save();  
    }  
}
```

```
}
}
```

Output: Reading, Writing और Saving अलग lines में।

9.4 Class Implements Multiple Interfaces

```
class Device implements Readable, Writable {
    public void read() { }
    public void write() { }
}
```

यदि दोनों interfaces compatible signature की एक abstract method declare करती हैं, तो एक suitable public implementation दोनों contracts पूरा कर सकती है। Incompatible return requirements वाली methods को एक class में implement करना संभव नहीं हो सकता।

9.5 Default Method Conflict

```
interface A {
    default void show() {
        System.out.println("A");
    }
}
interface B {
    default void show() {
        System.out.println("B");
    }
}
class C implements A, B {
    @Override
    public void show() {
        A.super.show();
        B.super.show();
    }
}
public class DefaultConflictDemo {
    public static void main(String[] args) {
        new C().show();
    }
}
```

Output: A फिर B। Unrelated inherited defaults की ambiguity class explicit override से resolve करती है।

9.6 Default Resolution के Basic Principles

- Class hierarchy की applicable concrete instance implementation generally interface default से priority लेती है।
- More-specific interface का override less-specific default से priority पा सकता है।
- Unrelated applicable defaults conflict करें, तो explicit resolution चाहिए।

- Access और abstract-method requirements फिर भी लागू रहती हैं।

इन principles को method signatures और complete hierarchy के संदर्भ में लागू करें; केवल interface names देखकर निर्णय न लें।

9.7 Class के भीतर Nested Interface

```
class Device {
    public interface Listener {
        void onEvent(String message);
    }
}
class ConsoleListener implements Device.Listener {
    public void onEvent(String message) {
        System.out.println("Event: " + message);
    }
}
public class NestedInterfaceDemo {
    public static void main(String[] args) {
        Device.Listener listener = new ConsoleListener();
        listener.onEvent("Started");
    }
}
```

Output: Event: Started

Member interface implicitly static होती है। Device.Listener उपयोग करने के लिए Device object बनाना आवश्यक नहीं है। Nested interface की accessibility declaration context पर निर्भर है।

9.8 Interface के भीतर Interface

```
interface Service {
    interface Callback {
        void completed();
    }
}
```

Service.Callback member type implicitly public static है। Callback implementations asynchronous/event-style APIs में उपयोगी हो सकती हैं।

9.9 Importance और Design

Nested interface संबंधित contract को enclosing API के साथ रख सकती है। Top-level reusable contract हो, तो separate interface अधिक स्पष्ट हो सकती है। Interface inheritance specialization का meaningful relationship दिखाए; unrelated capabilities के लिए multiple implementation भी पर्याप्त हो सकती है।

9.10 Quick Revision और अभ्यास

Interface extends one or more interfaces। Class implements interface। Default conflict explicit override से resolve करें। Nested interface outer object से bound inner instance type नहीं है।

अभ्यास: Printable और Exportable से ReportService interface बनाएँ। दो defaults का conflict resolve करें। Button.Listener nested interface का example लिखें।

MCQs: 1. Interface-interface keyword: A. implements B. extends C. new D. package 2. Interface कितनी interfaces extend कर सकती है? A. केवल एक B. अनेक C. कोई नहीं D. केवल Object 3. Member interface: A. हमेशा inner object-bound B. implicitly static C. constructor रखती है D. final method ही रखती है 4. Unrelated defaults conflict: A. Random selection B. Explicit resolution चाहिए C. दोनों automatic D. पहला alphabetically

उत्तर: 1-B, 2-B, 3-B, 4-B.

UNIT II / CHAPTER 10

अध्याय 10: Packages का परिचय एवं निर्माण

Definition, Naming Rules एवं User-Defined Packages

10.1 Learning Objectives

Package का अर्थ, built-in packages, naming conventions, package declaration और source directory organization समझेंगे।

10.2 Package क्या है?

Package related types को namespace में व्यवस्थित करता है। Class names के conflicts कम करने और package-level access नियंत्रित करने में उपयोगी है।

एक application में college.students.Student और college.staff.Student जैसे fully qualified names अलग types बता सकते हैं। समान simple name का उपयोग सावधानी से करना चाहिए।

10.3 Built-In Packages

Package	प्रमुख उदाहरण
java.lang	Object, String, System, Math
java.util	Scanner, Arrays, ArrayList
java.io	Input/output classes
java.nio.file	Paths और file operations
java.time	Date/time classes
java.net	Networking

Java.lang automatically imported है। Import automatically subpackages तक नहीं फैलता।

10.4 Package Declaration

```
package college.tools;

public class Calculator {
    public static int add(int a, int b) {
        return a + b;
    }
}
```

Comments के बाद package declaration और उसके बाद imports तथा type declarations रखी जाती हैं। Ordinary compilation unit में एक package declaration होती है।

10.5 Naming Rules और Conventions

Package components valid identifiers होने चाहिए; keywords उपयोग नहीं कर सकते। Lowercase नाम convention हैं। Organization अक्सर reverse-domain format उपयोग करती है: com.example.education।

नाम	स्थिति
college.tools	उपयुक्त conventional name
com.example.app	उपयुक्त reverse-domain style
college.2tools	Invalid: component digit से शुरू
college.class	Invalid: keyword
College.Tools	संभव identifier structure, convention के विपरीत

10.6 Package और Directory

Source organization convention:

File	Path
Calculator.java	src/college/tools/Calculator.java
App.java	src/college/app/App.java

Package language namespace है; directory runtime/build organization में सहायता करती है। Compiler को file directly देने के कुछ workflows में source कहीं और भी हो सकती है, लेकिन consistent matching hierarchy रखना अच्छा अभ्यास है।

10.7 दो Packages का Complete Example

File: src/college/tools/Calculator.java

```
package college.tools;
public class Calculator {
    public static int add(int a, int b) {
        return a + b;
    }
}
```

File: src/college/app/App.java

```
package college.app;
import college.tools.Calculator;
```

```
public class App {  
    public static void main(String[] args) {  
        System.out.println(Calculator.add(10, 20));  
    }  
}
```

Compile command एक line में लिखें:

```
javac -d out src/college/tools/Calculator.java src/college/app/App.java  
java -cp out college.app.App
```

Output: 30

10.8 Public Types और Filenames

Normal file-based Java compilation में public top-level class का filename उसी नाम से होता है। Calculator public है, इसलिए Calculator.java रखें। दूसरे package से उसका public add method accessible है। Class public हो लेकिन method package-private हो, तो दूसरे package से call नहीं कर पाएँगे।

10.9 Unnamed Package

Package declaration न हो, तो type unnamed package में होती है। यह छोटे introductory examples में सरल है। बड़े projects में named packages उपयोग करें। Named package source से unnamed-package types को सामान्य import करके उपयोग नहीं किया जाता।

10.10 Subpackages का भ्रम

College और college.tools अलग packages हैं। Names में prefix साझा होने से same-package access नहीं मिलता। Package-private class college में हो, तो college.tools से automatically accessible नहीं है।

10.11 Revision एवं अभ्यास

Packages namespace हैं। Fully qualified name package plus class name है। Matching source folders project management सरल करते हैं। Package-private और public access में अंतर रखें।

अभ्यास: College.geometry में Rectangle और Circle बनाएँ। College.app में area results दिखाएँ।

MCQs: 1. Package statement कहाँ? A. Types के बाद B. Imports/types से पहले C. Method में D. Constructor में 2. Package name convention: A. Lowercase B. हमेशा uppercase C. केवल digits D. spaces 3. Parent/subpackage package-private access: A. Automatic B. Automatic नहीं C. हमेशा public D. OS पर निर्भर

उत्तर: 1-B, 2-A, 3-B.

UNIT II / CHAPTER 11

अध्याय 11: Classpath, Imports एवं Access Protection

Classes खोजना और Visibility नियंत्रित करना

11.1 Learning Objectives

PATH और classpath का अंतर, classpath root, import forms तथा public/private/protected/package-private access समझेंगे। Cross-package inheritance के विशेष protected rule का अभ्यास करेंगे।

11.2 PATH बनाम Classpath

व्यवस्था	क्या खोजती है?	उदाहरण
PATH	Executable commands	java, javac
Classpath	Application classes/resources	Compiled directories और JARs

Classpath classpath-mode programs में class loading locations बताता है। Modules का module-path अलग concept है, जो इस introductory unit से बाहर है।

11.3 Classpath Root

यदि compiled file out/college/app/App.class है और type college.app.App है, तो root out है।

```
java -cp out college.app.App
```

गलत root out/college/app देना class lookup fail कर सकता है। Default configurations में current directory classpath होती है, लेकिन explicit -cp prior setting को replace करता है। आवश्यकता हो तो dot शामिल करें।

11.4 Multiple Entries

Windows:

```
java -cp "out;lib/helper.jar" college.app.App
```

Linux/macOS:

```
java -cp "out:lib/helper.jar" college.app.App
```

Classpath separator Windows में semicolon और Linux/macOS में colon है। Paths में spaces हों तो proper quoting करें। Helper.jar तभी लिखें जब वह actual dependency मौजूद हो।

11.5 Imports के प्रकार

```
import java.util.Scanner;           // Single-type import
import java.util.*;                // On-demand type import
import static java.lang.Math.PI;   // Static member import
```

Wildcard subpackages import नहीं करती। Import runtime में सारी classes load करने का निर्देश नहीं है। Two types का same simple name conflict करे, तो fully qualified name उपयोग करें।

11.6 Access Modifiers की तुलना

यह table सामान्य members के लिए है; enclosing type भी accessible होना चाहिए।

Modifier	Same class	Same package	Other-package subclass	Other-package unrelated
public	हाँ	हाँ	हाँ	हाँ
protected	हाँ	हाँ	नियमों के अनुसार	नहीं
No modifier	हाँ	हाँ	नहीं	नहीं
private	हाँ	नहीं*	नहीं*	नहीं

*Private nested-type access के nest-related नियम हैं। सामान्य separate top-level classes के लिए ऊपर का सरल distinction उपयोग करें।

Top-level ordinary classes public या package-private हो सकती हैं; top-level private/protected class declaration नहीं लिखते।

11.7 Cross-Package protected का विशेष नियम

Superclass का protected instance member दूसरे package में subclass implementation से accessible हो सकता है। लेकिन subclass के बाहर मनमाने superclass object के माध्यम से access नहीं मिलता।

File: src/p1/Parent.java

```
package p1;
public class Parent {
    protected int value = 10;
}
```

File: src/p2/Child.java

```
package p2;
import p1.Parent;
public class Child extends Parent {
```

```

    public void show(Child other) {
        System.out.println(this.value);
        System.out.println(other.value);
    }
    // void invalid(Parent parent) {
    //     System.out.println(parent.value); // Not allowed here
    // }
}

```

Protected cross-package instance access के लिए qualifier subclass type या उसके subtype के अनुरूप होना चाहिए। Protected को simply “subclass कहीं भी किसी Parent object को access कर सकती है” न समझें।

11.8 Private State और Public Behavior

```

package college.accounts;
public class Account {
    private double balance;
    public Account(double balance) {
        this.balance = balance;
    }
    public double getBalance() {
        return balance;
    }
}

```

Client balance सीधे नहीं बदल सकता, लेकिन public operation उपयोग कर सकता है। Getters और setters design के अनुसार दें; प्रत्येक field के लिए unrestricted setter आवश्यक नहीं है।

11.9 Executable JAR और Classpath

```

jar cfe CollegeApp.jar college.app.App -C out .
java -jar CollegeApp.jar

```

Executable JAR बनाते समय entry-point class बताई गई है। Dependency archives automatically merge नहीं होतीं। -jar mode में user -cp argument को regular class launch की तरह उपयोग नहीं किया जाता; required dependencies को suitable manifest/setup से व्यवस्थित करें।

11.10 Common Errors

Error	जाँच
Could not find main class	Classpath root और fully qualified name
Package does not exist	Source/dependency उपलब्धता
Member not accessible	Modifier और package relationship
Same simple name ambiguity	Explicit qualification

Error	जाँच
NoClassDefFoundError	Required runtime definition/dependency

11.11 Revision एवं अभ्यास

PATH commands ढूँढ़ता है; classpath application types ढूँढ़ता है। Import naming सहायता है। Package hierarchy automatic access hierarchy नहीं है। Protected qualifier rule cross-package access में महत्वपूर्ण है।

अभ्यास: Two packages में public और protected examples बनाएँ। Private/package-private failures को comments में explain करें। Out folder और JAR से application launch करें।

MCQs: 1. Windows classpath separator: A. : B. ; C. / D. , 2. Classpath root for out/p/App.class: A. out B. out/p C. App.class D. p 3. Top-level private class: A. Valid ordinary declaration B. Invalid C. आवश्यक D. Interface ही है 4. Imports subpackages: A. Always B. Automatically नहीं C. केवल java.util D. OS-dependent

उत्तर: 1-B, 2-A, 3-B, 4-B.

UNIT II / CHAPTER 12

अध्याय 12: Revision एवं Question Bank

Solved Outputs, MCQs एवं Mini Project

12.1 Unit II Revision Map

Topic	मुख्य बिंदु
Inheritance	is-a; extends
Class parents	एक direct superclass
Interfaces	Multiple contracts संभव
Constructors	Inherited नहीं; chained initialization
Overloading	Different parameter lists; compile-time selection
Overriding	Compatible instance signature; runtime dispatch
Abstract class	Direct instance नहीं; constructor संभव
final variable	Reassignment restriction
final method	Override restriction
final class	Extension restriction
Interface fields	public static final
Nested interface	Implicitly static member type
Package	Namespace
Classpath	Classes/resources की locations
Access	Modifier plus enclosing type/package rules

12.2 Important Differences

Pair	अंतर
Inheritance / composition	is-a / has-a
Overloading / overriding	Signatures का विकल्प / specialized implementation

Pair	अंतर
Constructor invocation / inheritance	Initialization chain / members का subtype संबंध
Static method / instance method	Class-resolved / overridable dispatch possible
Abstract class / interface	Shared base state / multiple contracts
Final reference / immutable object	Reassignment रोकना / content mutation रोकना
PATH / classpath	Executables / application definitions
Import / loading	Source name resolution / runtime availability

12.3 Solved Output Questions

Q1: Runtime dispatch

```
class A { void show() { System.out.println("A"); } }
class B extends A {
    @Override void show() { System.out.println("B"); }
}
// Inside main:
// A value = new B();
// value.show();
```

उत्तर: B Actual object की overridden implementation चुनी जाती है।

Q2: Constructor chain

```
class A { A() { System.out.print("A "); } }
class B extends A { B() { System.out.print("B "); } }
// Inside main: new B();
```

उत्तर: A B Parent body पहले।

Q3: Field hiding

```
class A { int number = 10; }
class B extends A { int number = 20; }
// Inside main:
// A value = new B();
// System.out.println(value.number);
```

उत्तर: 10 Fields overridden instance methods जैसे dispatch नहीं होतीं।

Q4: Overload selection

```
static void show(Object value) { System.out.println("Object"); }
static void show(String value) { System.out.println("String"); }
```

```
// Object value = "Java";
// show(value);
```

उत्तर: Object | Declared argument type से overload selection।

Q5: Final array

```
final int[] marks = {70, 80};
marks[0] = 95;
System.out.println(marks[0]);
```

उत्तर: 95 | Reference final है; element mutable है।

Q6: Abstract constructor

```
abstract class A {
    A() { System.out.print("A "); }
}
class B extends A {
    B() { System.out.print("B "); }
}
// new B();
```

उत्तर: A B | Abstract base constructor चलता है।

12.4 MCQ Question Bank

1. Class inheritance keyword? A. implements B. extends C. import D. package
2. Inheritance relationship? A. is-a B. has-a C. contains-only D. none
3. Constructors inherited? A. हाँ B. नहीं C. केवल public D. केवल private
4. One parent-many children? A. Hierarchical B. Multilevel C. Multiple direct classes D. None
5. Class की direct superclass? A. अधिकतम एक B. Unlimited C. हमेशा दो D. तीन
6. Class अनेक interfaces implement कर सकती है? A. हाँ B. नहीं C. केवल final D. केवल private
7. super(args) किसे call करता है? A. Direct parent constructor B. Interface constructor C. Package D. JVM
8. A-B-C constructor bodies order? A. C-B-A B. A-B-C C. B-A-C D. Random
9. Overloading में क्या अलग चाहिए? A. केवल return B. Parameter list C. केवल name spelling case same D. केवल variable names
10. Return type alone overload? A. पर्याप्त B. पर्याप्त नहीं C. हमेशा valid D. केवल void
11. Overload selection? A. Compile time B. GC time C. हमेशा actual runtime object D. JAR creation
12. Overridden instance dispatch? A. Runtime object B. केवल reference name C. Filename D. Import order
13. @Override का लाभ? A. Intent checking B. Object deletion C. Package creation D. Constructor inheritance
14. Public method override का private access? A. Allowed B. Invalid C. Required D. केवल Windows

15. Static methods? A. Instance override B. Hide हो सकती हैं C. Abstract हमेशा D. Constructor
16. Final method? A. Override नहीं B. Call नहीं C. Inherit कभी नहीं D. हमेशा private
17. Final class? A. Extend नहीं B. Object नहीं C. Interface नहीं D. Methods नहीं
18. Abstract class direct instance? A. हाँ B. नहीं C. केवल protected D. केवल no-arg
19. Abstract class constructor? A. संभव B. नहीं C. हमेशा abstract D. static आवश्यक
20. Abstract method body? A. आवश्यक B. नहीं C. केवल empty braces D. हमेशा return
21. Interface fields? A. public static final B. private instance C. protected D. transient always
22. Class-interface keyword? A. extends B. implements C. package D. import
23. Interface-interface keyword? A. extends B. implements C. new D. final
24. Interface constructor? A. नहीं B. हाँ C. हमेशा default D. inherited
25. Default interface methods introduced? A. Java 8 B. Java 1 only C. Java 5 D. Java 2
26. Private interface helpers introduced? A. Java 9 B. Java 1 C. Java 5 D. Java 2
27. Member interface? A. implicitly static B. Outer instance आवश्यक C. Constructor रखती है D. primitive
28. Package naming convention? A. lowercase B. uppercase only C. digits only D. spaces
29. java.lang import? A. Automatic B. कभी नहीं C. केवल JAR D. केवल Android
30. Wildcard import subpackages? A. नहीं B. सभी C. केवल first D. Random
31. PATH खोजता है? A. Executables B. Objects C. Comments D. केवल classes
32. Classpath root for out/p/App.class? A. out B. out/p C. p D. App.class
33. Windows classpath separator? A. semicolon B. colon C. comma D. slash
34. Linux classpath separator? A. colon B. semicolon C. comma D. backslash
35. Top-level private class? A. Invalid B. Always valid C. Required D. Package only
36. No-modifier member access? A. Same package B. हर package C. केवल subclasses हर जगह D. हमेशा private
37. Final reference object mutable हो सकता है? A. हाँ B. नहीं C. केवल null D. केवल interface
38. Abstract final class? A. Invalid B. Valid C. Required D. Default
39. Interface static method सामान्य call? A. InterfaceName.method() B. super() C. new method D. import()
40. Two unrelated defaults conflict? A. Explicit override resolution B. Random C. पहला alphabetic D. कोई check नहीं

12.5 Answer Key

प्रश्न	उत्तर	प्रश्न	उत्तर	प्रश्न	उत्तर	प्रश्न	उत्तर
1	B	11	A	21	A	31	A
2	A	12	A	22	B	32	A
3	B	13	A	23	A	33	A

प्रश्न	उत्तर	प्रश्न	उत्तर	प्रश्न	उत्तर	प्रश्न	उत्तर
4	A	14	B	24	A	34	A
5	A	15	B	25	A	35	A
6	A	16	A	26	A	36	A
7	A	17	A	27	A	37	A
8	B	18	B	28	A	38	A
9	B	19	A	29	A	39	A
10	B	20	B	30	A	40	A

12.6 Short-Answer Questions

1. Superclass और subclass की परिभाषा दीजिए।
2. Constructors inherited क्यों नहीं कहलाते?
3. This(...) और super(...) का अंतर लिखिए।
4. Overloading और overriding में अंतर लिखिए।
5. Covariant return type क्या है?
6. Abstract class में constructor का उपयोग क्या है?
7. Final reference और immutable object में अंतर बताइए।
8. Interface fields के implicit modifiers लिखिए।
9. Nested interface की creation/use समझाइए।
10. PATH तथा classpath का अंतर लिखिए।

12.7 Long-Answer Questions

1. Inheritance के प्रकार diagrams और programs सहित समझाइए।
2. Multilevel constructor chain का execution order समझाइए।
3. Runtime dispatch और field hiding की तुलना कीजिए।
4. Abstract class तथा interface की विस्तृत तुलना कीजिए।
5. Final variables, methods और classes उदाहरण सहित समझाइए।
6. Interface inheritance और default conflicts समझाइए।
7. Packages बनाने, compile करने और चलाने के चरण लिखिए।
8. Access modifiers तथा cross-package protected rule समझाइए।

12.8 Mini Project: College Payroll

यह project abstract class, inheritance, constructors, interface, overriding, final fields और polymorphic references को एक साथ उपयोग करता है। Teaching example में amounts double हैं; exact financial decimal arithmetic के लिए suitable decimal representation चुननी चाहिए।

Save as PayrollApp.java:

```
interface Payable {
    double amount();
}

abstract class Employee implements Payable {
    private final String name;

    Employee(String name) {
        this.name = name;
    }

    final String getName() {
        return name;
    }

    void display() {
        System.out.printf("%s: %.2f%n", name, amount());
    }
}

class PermanentEmployee extends Employee {
    private final double basic;
    private final double allowance;

    PermanentEmployee(String name, double basic,
                       double allowance) {
        super(name);
        this.basic = basic;
        this.allowance = allowance;
    }

    @Override
    public double amount() {
        return basic + allowance;
    }
}

class ContractEmployee extends Employee {
    private final int hours;
    private final double hourlyRate;

    ContractEmployee(String name, int hours,
                     double hourlyRate) {
        super(name);
        this.hours = hours;
        this.hourlyRate = hourlyRate;
    }

    @Override
    public double amount() {
        return hours * hourlyRate;
    }
}
```

```

}

public class PayrollApp {
    public static void main(String[] args) {
        Employee[] employees = {
            new PermanentEmployee("Amit", 30000, 5000),
            new ContractEmployee("Neha", 80, 250)
        };

        double total = 0;
        for (Employee employee : employees) {
            employee.display();
            total += employee.amount();
        }
        System.out.printf("Total: %.2f%n", total);
    }
}

```

12.9 Commands और Expected Output

```

javac PayrollApp.java
java PayrollApp

```

```

Amit: 35000.00
Neha: 20000.00
Total: 55000.00

```

Locale-specific formatting में decimal separator बदल सकता है। Calculation: Permanent = 30000+5000; Contract = 80×250; total = 55000।

12.10 Project Extensions

- Negative hours/rates और null names की validation जोड़ें।
- Classes को college.payroll package में अलग files में रखें।
- ReportWriter nested interface बनाएँ।
- Overloaded constructor से default allowance दें।
- PayrollApp की executable JAR बनाएँ।
- Abstract base के बजाय Payable reference array से calculations करें।

12.11 Revision Checklist

क्या आप superclass reference से overridden method का output बता सकते हैं? क्या overload selection में declared type का महत्त्व समझते हैं? क्या constructors और inherited members में अंतर स्पष्ट है? क्या protected access को package context सहित जाँच सकते हैं? क्या packaged main class के लिए सही classpath root लिख सकते हैं? प्रत्येक अस्पष्ट बिंदु का छोटा program बनाकर परिणाम देखें।