Module 3: Interfaces and Packages

Java OOP — Semester 3

Based on NDU Curriculum DCSA250059

PART A: INTERFACES

1. Introduction to Interfaces

An **interface** is a blueprint of a class containing abstract methods (methods without body). It defines a **contract** that implementing classes must follow.

Key Characteristics

- All methods are public abstract by default (except default and static methods)
- All variables are public static final by default (constants)
- Cannot be instantiated — no constructors
- Supports multiple inheritance

Syntax

```
interface InterfaceName {
    // Constants (public static final)
    int MAX_VALUE = 100;
    String DEFAULT_NAME = "Unknown";

    // Abstract methods (public abstract)
    void method1();
    int method2(String param);

    // Default method (Java 8+)
    default void defaultMethod() {
        System.out.println("Default implementation");
    }

    // Static method (Java 8+)
    static void staticMethod() {
        System.out.println("Static method");
    }
}
```

2. Defining and Implementing Interfaces

A class uses the implements keyword to fulfill an interface contract and must override all abstract methods.

```
// Defining an interface
interface Drawable {
    void draw();            // abstract method
```

```
    void resize(int size);  // abstract method
}

// Implementing the interface
class Circle implements Drawable {
    private int radius;
    Circle(int radius) { this.radius = radius; }

    @Override
    public void draw() {
        System.out.println("Drawing circle with radius: " + radius);
    }

    @Override
    public void resize(int size) {
        radius = size;
        System.out.println("Circle resized to: " + radius);
    }
}

public class InterfaceDemo {
    public static void main(String[] args) {
        Drawable d1 = new Circle(5);  // Polymorphism
        d1.draw();
        d1.resize(10);
    }
}
```

3. Multiple Interfaces Implementation

Java doesn't support multiple inheritance with classes, but **supports multiple interfaces**. A class can implement as many interfaces as needed, separated by commas.

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

class Document implements Printable, Showable, Transmittable {
    private String content;
    Document(String content) { this.content = content; }

    @Override public void print()    { System.out.println("Printing: "    +
content); }
    @Override public void show()     { System.out.println("Showing: "     +
content); }
    @Override public void transmit() { System.out.println("Transmitting: "+
content); }
}

// Interface polymorphism
Document doc = new Document("Hello World");
Printable p = doc;
p.print();  // Only print() accessible via this reference
```

4. Extending Interfaces

Interfaces can extend other interfaces (and even multiple interfaces simultaneously), unlike classes.

```
interface Animal  { void eat(); void sleep(); }
interface Mammal extends Animal { void giveBirth(); void breathe(); }

interface Flying  { void fly(); }
interface Swimming{ void swim(); }

// Extending multiple interfaces
interface Amphibian extends Flying, Swimming {
    void liveOnLand();
    void liveInWater();
}

class Bat implements Mammal {
    @Override public void eat()       { System.out.println("Bat eating insects"); }
    @Override public void sleep()     { System.out.println("Bat sleeping upside
down"); }
    @Override public void giveBirth() { System.out.println("Bat giving birth"); }
    @Override public void breathe()   { System.out.println("Bat breathing"); }
}
```

5. Interface Variables (Constants)

All variables in an interface are implicitly **public, static, and final**. They must be initialized at declaration and cannot be changed.

```
interface Constants {
    int MAX_SIZE = 100;               // public static final
    String APPLICATION_NAME = "MyApp"; // public static final
    double VERSION = 1.0;             // public static final
}

interface Config {
    String DB_URL  = "jdbc:mysql://localhost:3306/mydb";
    String DB_USER = "root";
    int    TIMEOUT = 30;
}

class DatabaseManager implements Config {
    void connect() {
        System.out.println("Connecting to: " + DB_URL);
        // DB_URL = "newurl";  // ERROR - constants are final
    }
}

// Direct access: Constants.MAX_SIZE
System.out.println("Max: " + Constants.MAX_SIZE);
```

6. Default Methods (Java 8+)

Default methods allow adding new methods to interfaces without breaking existing implementations. Classes may optionally override them.

```
interface Vehicle {
    void start();
    void stop();

    default void honk() {
        System.out.println("Vehicle honking: Beep beep!");
    }
}

class Car implements Vehicle {
    @Override public void start() { System.out.println("Car engine started"); }
    @Override public void stop()  { System.out.println("Car stopped"); }

    @Override
    public void honk() {                          // Override default
        System.out.println("Car honking: Peeep!");
    }
}

class Bike implements Vehicle {
    @Override public void start() { System.out.println("Bike kick-started"); }
    @Override public void stop()  { System.out.println("Bike stopped"); }
    // Uses default honk()
}
```

Resolving the Diamond Problem

When two interfaces have the same default method name, the implementing class must override it to resolve the conflict.

```
interface A { default void display() { System.out.println("A's display"); } }
interface B { default void display() { System.out.println("B's display"); } }

class C implements A, B {
    @Override
    public void display() {
        System.out.println("C resolving conflict");
        A.super.display();  // Call A's default
        B.super.display();  // Call B's default
    }
}
```

7. Static Methods in Interfaces (Java 8+)

Static methods in interfaces are NOT inherited by implementing classes and must be called through the interface name.

```
interface MathUtils {
    static int add(int a, int b)      { return a + b; }
    static int multiply(int a, int b) { return a * b; }
```

```
    int subtract(int a, int b);  // abstract
}

// Call via interface name (not via instance)
System.out.println(MathUtils.add(10, 5));       // 15
System.out.println(MathUtils.multiply(10, 5));  // 50

Calculator calc = new Calculator();
System.out.println(calc.subtract(10, 5));       // 5
// calc.add(10, 5);  // ERROR - static not inherited
```

8. Private Methods in Interfaces (Java 9+)

Private methods allow code reuse within default or static methods inside the interface, without exposing the helper logic to implementing classes.

```
interface DataProcessor {
    default void processString(String data) {
        String validated = validate(data);        // uses private helper
        String formatted = format(validated);     // uses private static helper
        System.out.println("Processed: " + formatted);
    }

    private String validate(String data) {       // private instance method
        return (data == null || data.trim().isEmpty()) ? "EMPTY" : data.trim();
    }

    private static String format(String data) {  // private static method
        return data.toUpperCase();
    }
}
```

9. Functional Interfaces (Java 8+)

An interface with **exactly one abstract method** is a **functional interface**. It can be used with **lambda expressions** for concise implementations.

```
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);  // Single abstract method
    // default / static methods are allowed too
}

// Anonymous class (traditional)
Calculator add = new Calculator() {
    @Override public int calculate(int a, int b) { return a + b; }
};

// Lambda expressions (Java 8+)
Calculator subtract = (a, b) -> a - b;
Calculator multiply = (a, b) -> a * b;
Calculator divide   = (a, b) -> a / b;
```

```
System.out.println(add.calculate(10, 5));       // 15
System.out.println(subtract.calculate(10, 5));  // 5
```

Common Built-in Functional Interfaces (java.util.function)

Interface	Input(s)	Return	Example
Predicate<T>	T	boolean	n -> n % 2 == 0
Function<T,R>	T	R	s -> s.length()
Consumer<T>	T	void	s -> System.out.println(s)
Supplier<T>	none	T	() -> Math.random()
BiFunction<T,U,R>	T, U	R	(a,b) -> a > b ? a : b
BiPredicate<T,U>	T, U	boolean	(a,b) -> a > b

10. Marker Interface

A marker interface has **no methods** — it serves as metadata/tag, signalling to the JVM or frameworks that a class has certain capabilities.

```
interface Auditable { }        // custom marker — no methods
// Built-in examples: java.io.Serializable, java.lang.Cloneable

class Employee implements Auditable, java.io.Serializable {
    String name; int id;
}

Employee emp = new Employee();
if (emp instanceof Auditable)         System.out.println("Employee is auditable");
if (emp instanceof java.io.Serializable) System.out.println("Can be serialized");
```

11. Interface vs Abstract Class

Feature	Interface	Abstract Class
Multiple inheritance	Yes	No
Instance variables	No (constants only)	Yes
Constructors	No	Yes
Access modifiers	Public (default)	All modifiers
Method implementation	default/static (Java 8+)	Concrete methods allowed
When to use	Define contract/capability	Share common code

Combined Example

```
abstract class Shape {
    protected String color;
    Shape(String color) { this.color = color; }
    abstract double area();
    void displayColor() { System.out.println("Color: " + color); }
}

interface Drawable  { void draw(); }
interface Resizable { void resize(int factor); }

class Circle extends Shape implements Drawable, Resizable {
    private double radius;
    Circle(String color, double radius) { super(color); this.radius = radius; }

    @Override double area()            { return Math.PI * radius * radius; }
    @Override public void draw()       { System.out.println("Drawing circle"); }
    @Override public void resize(int f){ radius *= f; }
}
```

PART B: PACKAGES

12. Introduction to Packages

A **package** is a namespace that groups related classes, interfaces, and sub-packages.

Benefits

- Organization — groups related classes together
- Access protection — package-private access level
- Name conflict prevention — same class name in different packages
- Code reusability — easy to share and distribute as JARs

Types of Packages

Type	Examples	Notes
Built-in (Java API)	java.lang, java.util, java.io	java.lang is auto-imported
User-defined	com.company.project.module	Follow reverse-domain naming

13. Creating Packages

Naming Convention

```
// Reverse domain name pattern:
```

```
com.companyname.projectname.modulename

// Examples:
com.google.search.engine
org.apache.commons.lang
edu.university.cs.program
```

Package Declaration — File: com/mycompany/student/Student.java

```
package com.mycompany.student;  // MUST be the very first statement

import java.util.Date;

public class Student {
    private int rollNo;
    private String name;
    private Date admissionDate;

    public Student(int rollNo, String name) {
        this.rollNo = rollNo;
        this.name = name;
        this.admissionDate = new Date();
    }

    public void display() {
        System.out.println("Roll: " + rollNo + ", Name: " + name);
    }

    void internalMethod() { /* package-private */ }
}
```

Directory Structure

```
src/
└── com/
    └── mycompany/
        ├── student/
        │   ├── Student.java
        │   └── GraduateStudent.java
        └── teacher/
            └── Teacher.java
```

Compilation Commands

```
# Compile from src directory
javac com/mycompany/student/*.java com/mycompany/teacher/*.java

# Run
java com.mycompany.student.Student
```

14. Using Packages (Import Styles)

```
// Method 1: Import a specific class
import com.mycompany.student.Student;

// Method 2: Import entire package (on-demand)
import com.mycompany.student.*;
import com.mycompany.teacher.*;

// Method 3: Fully qualified name — no import needed
com.mycompany.teacher.Teacher t =
    new com.mycompany.teacher.Teacher(201, "Prof. Smith", "Physics");
```

15. Java API Packages — Quick Reference

Package	Description	Key Classes
java.lang	Core (auto-imported)	String, Math, System, Object, Thread
java.util	Utility classes	ArrayList, HashMap, Scanner, Collections
java.io	Input / Output	File, BufferedReader, PrintWriter
java.net	Networking	Socket, ServerSocket, URL
java.awt	GUI (legacy)	Frame, Button, Panel, Graphics
javax.swing	GUI (modern)	JFrame, JButton, JTable
java.sql	Database	Connection, Statement, ResultSet
java.time	Date/Time (Java 8+)	LocalDate, LocalDateTime, Duration

```
import java.util.*;
import java.time.*;
import java.time.format.DateTimeFormatter;

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
Collections.sort(names);

Map<String, Integer> scores = new HashMap<>();
scores.put("Alice", 95);

LocalDate today = LocalDate.now();
DateTimeFormatter fmt = DateTimeFormatter.ofPattern("yyyy-MM-dd");
System.out.println("Today: " + today.format(fmt));
```

16. Static Import

Static import lets you access static members (fields and methods) without prefixing the class name.

```
import static java.lang.Math.*;
```

```
import static java.lang.System.out;
import static java.util.Collections.*;

out.println("PI: "  + PI);           // instead of Math.PI
out.println(sqrt(16));               // instead of Math.sqrt(16)
out.println(pow(2, 3));              // instead of Math.pow(2, 3)

List<String> list = new ArrayList<>();
addAll(list, "A", "B", "C");
shuffle(list);
sort(list);
out.println(list);
```

17. Access Protection in Packages

Modifier	Same Class	Same Package	Subclass	World
private	✓	✗	✗	✗
default (none)	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

```
// com.mycompany.access.Parent
public class Parent {
    private int   privateVar   = 1;  // class only
    int           defaultVar   = 2;  // same package
    protected int protectedVar = 3;  // package + subclasses
    public int    publicVar    = 4;  // everywhere
}

// com.mycompany.subclass.Child  (different package, extends Parent)
public class Child extends Parent {
    void test(Child c) {
        // c.privateVar;   // ✗ private
        // c.defaultVar;   // ✗ different package
        c.protectedVar;    // ✓ inherited
        c.publicVar;       // ✓ public
    }
}
```

18. Creating and Using JAR Files

A JAR (Java Archive) bundles multiple .class files and resources into a single portable file.

```
# Create JAR
jar cf mylibrary.jar com/mycompany/**/*.class

# Create executable JAR with manifest
jar cfm mylibrary.jar manifest.txt com/mycompany/**/*.class
```

```
# manifest.txt contents:
# Manifest-Version: 1.0
# Main-Class: com.mycompany.MainClass

# View contents
jar tf mylibrary.jar

# Use JAR in classpath (Windows ; Linux :)
javac -cp mylibrary.jar MyProgram.java
java  -cp .;mylibrary.jar MyProgram   # Windows
java  -cp .:mylibrary.jar MyProgram   # Linux/Mac

# Run executable JAR
java -jar mylibrary.jar
```

19. Complete Example — Banking System

A multi-package banking application demonstrating packages, abstract classes, interfaces, and access control.

Package Structure

```
com.bank.model   — Account (abstract), SavingsAccount
com.bank.service — BankService
com.bank.main    — BankingApplication (entry point)
```

com/bank/model/Account.java

```
package com.bank.model;

public abstract class Account {
    private int accountNo;
    private String holderName;
    protected double balance;

    public Account(int accountNo, String holderName, double initialBalance) {
        this.accountNo   = accountNo;
        this.holderName  = holderName;
        this.balance     = initialBalance;
    }

    public int    getAccountNo()   { return accountNo; }
    public String getHolderName()  { return holderName; }
    public double getBalance()     { return balance; }

    public abstract void deposit(double amount);
    public abstract void withdraw(double amount);

    public void display() {
        System.out.printf("Account: %d | Holder: %s | Balance: %.2f%n",
                          accountNo, holderName, balance);
    }
```

```
}
```

com/bank/model/SavingsAccount.java

```
package com.bank.model;

public class SavingsAccount extends Account {
    private double interestRate;

    public SavingsAccount(int accountNo, String holderName,
                          double initialBalance, double interestRate) {
        super(accountNo, holderName, initialBalance);
        this.interestRate = interestRate;
    }

    @Override
    public void deposit(double amount) {
        if (amount > 0) { balance += amount; System.out.println("Deposited: " +
amount); }
    }

    @Override
    public void withdraw(double amount) {
        if (amount > 0 && balance >= amount) {
            balance -= amount; System.out.println("Withdrawn: " + amount);
        } else { System.out.println("Insufficient balance"); }
    }

    public void addInterest() {
        double interest = balance * interestRate / 100;
        balance += interest;
        System.out.println("Interest added: " + interest);
    }
}
```

com/bank/service/BankService.java

```
package com.bank.service;
import com.bank.model.Account;
import java.util.*;

public class BankService {
    private Map<Integer, Account> accounts = new HashMap<>();

    public void addAccount(Account account) {
        accounts.put(account.getAccountNo(), account);
    }

    public Account findAccount(int accountNo) { return accounts.get(accountNo); }

    public void transfer(int fromAcc, int toAcc, double amount) {
        Account from = accounts.get(fromAcc);
        Account to   = accounts.get(toAcc);
        if (from == null || to == null) { System.out.println("Invalid account");
return; }
        from.withdraw(amount);
        to.deposit(amount);
        System.out.println("Transfer completed");
```

```
    }

    public void displayAllAccounts() {
        accounts.values().forEach(Account::display);
    }
}
```

20. Quick Reference

Interface Syntax

```
interface MyInterface {
    int CONSTANT = 100;              // public static final
    void method1();                  // public abstract
    default void method2() { }      // default (Java 8+)
    static  void method3() { }      // static (Java 8+)
    private void method4() { }      // private (Java 9+)
}
```

Package Declaration Order

```
package com.company.pkg;  // 1st — package declaration
import java.util.*;       // 2nd — import statements
public class ClassName {} // 3rd — class definition
```

Common Import Patterns

```
import java.util.ArrayList;        // single class
import java.util.*;                // entire package
import static java.lang.Math.PI;   // static member
import static java.lang.Math.*;    // all static members
```

Key Points

- Interface — contract that classes must implement using the implements keyword
- Multiple inheritance — only possible with interfaces, not abstract classes
- Default methods (Java 8+) — allow adding methods without breaking existing implementations
- Functional interface — exactly one abstract method; eligible for lambda expressions
- Marker interface — no methods; serves as metadata/tagging (e.g. Serializable)
- Package — namespace that groups related classes; prevents name conflicts
- Access levels (widest to narrowest): public > protected > default > private
- Static import — access static members without prefixing the class name
- JAR file — bundles compiled .class files into a single distributable archive