//Name:Raiyan Riyaz Chandle

//Rollno:B-15

//Div:B

//PRN:2324000573

1. What is Interface in Java. Write Java Syntax for Interface. Write Java Program example to declare and implement Interface in Class.

Definition of an Interface in Java

An interface in Java is a reference type similar to a class but is designed to define a set of methods that implementing classes must adhere to. It can contain method signatures, constants, default methods, static methods, and nested types. However, an interface does not support instance fields or constructors. Interfaces enable abstraction and allow multiple inheritance in Java.

Syntax of an Interface

java
CopyEdit

```
interface InterfaceName {
    // Abstract methods
    void method1();
    void method2();

    // Default method
    default void defaultMethod() {
        // Method implementation
    }

    // Static method
    static void staticMethod() {
        // Method implementation
    }
}
```

Example Program

Defining an Interface

java
CopyEdit

```
interface Animal {
    void eat();
    void sleep();
}
```

Implementing the Interface in a Class

java
CopyEdit

```
class Dog implements Animal {
    public void eat() {
        System.out.println("Dog is eating.");
    }

    public void sleep() {
        System.out.println("Dog is sleeping.");
```

```
    }
}
```

Main Class to Test Implementation

```
java
CopyEdit
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();
        myDog.sleep();
    }
}
```

Explanation

- The Animal interface declares two abstract methods: eat() and sleep().
- The Dog class implements the Animal interface and provides concrete implementations for these methods.
- In the Main class, an instance of Dog is created, and the methods are called, producing the following output:

Output

```
csharp
CopyEdit
Dog is eating.
Dog is sleeping.
```

Diagram Representation

```
lua
CopyEdit
+------------------+
|    Animal        |  <--- Interface
|------------------|
| + eat()          |
| + sleep()        |
+------------------+
        ^
        |
        | Implements
        |
+------------------+
|      Dog         |
|------------------|
| + eat()          |
| + sleep()        |
+------------------+
```

2. What is relation between Interface and Class. Can single class implement two Interface? If yes write Java Syntax to how single class implements two Interfaces. Also write Java Program to show single class implements two interface and add logic to all methods.

Relationship Between an Interface and a Class

In Java, an interface defines a set of abstract methods that a class must implement, acting as a contract for behavior. While a class provides a blueprint for objects, including attributes and behaviors, an interface specifies only the "what" (method signatures) without dictating the "how" (implementation). This approach enhances code flexibility, maintainability, and scalability.

Implementing Multiple Interfaces

Java allows a class to implement multiple interfaces, enabling a form of multiple inheritance. Since Java does not support multiple inheritance through classes, this feature helps in designing modular and reusable code.

Syntax for Implementing Multiple Interfaces

```
java
CopyEdit
interface Interface1 {
    void method1();
}

interface Interface2 {
    void method2();
}

class MyClass implements Interface1, Interface2 {
    public void method1() {
        // Implementation of method1
    }

    public void method2() {
        // Implementation of method2
    }
}
```

Example Program

Defining the First Interface

```
java
CopyEdit
interface Animal {
    void eat();
}
```

Defining the Second Interface

```
java
CopyEdit
interface Pet {
    void play();
}
```

Implementing Both Interfaces in a Single Class

```
java
CopyEdit
class Dog implements Animal, Pet {
    public void eat() {
        System.out.println("Dog is eating.");
    }
```

```
    public void play() {
        System.out.println("Dog is playing.");
    }
}
```

Main Class to Test Implementation

java
CopyEdit
```
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();
        myDog.play();
    }
}
```

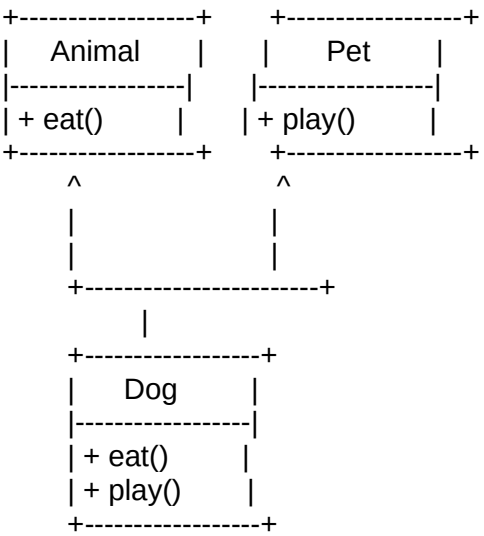
Explanation

- The Animal interface defines an abstract method eat().
- The Pet interface defines another abstract method play().
- The Dog class implements both interfaces and provides concrete implementations for the methods.
- In the Main class, an instance of Dog is created, and both eat() and play() methods are called, producing the following output:

Output

csharp
CopyEdit
```
Dog is eating.
Dog is playing.
```

Diagram Representation

lua
CopyEdit


3. List Down One Real-Time Example Where: a. Java Class Uses a Single Interface b. Java Class Uses Two Interfaces

a. Java Class Implementing a Single Interface

Real-World Example: Payment System

In a payment processing system, different payment methods can be implemented using a common interface. Each method follows a standard contract defined by the interface but implements its own logic.

Interface Definition

java
CopyEdit
```
interface Payment {
    void processPayment(double amount);
}
```

Class Implementing the Interface

java
CopyEdit
```
class CreditCardPayment implements Payment {
    public void processPayment(double amount) {
        System.out.println("Processing credit card payment of $" + amount);
    }
}
```

Explanation

- The Payment interface defines a method for processing payments.
- The CreditCardPayment class implements the Payment interface and provides a specific implementation for handling credit card transactions.

Diagram Representation

lua
CopyEdit
```
+---------------------+
|      Payment        | <--- Interface
|---------------------|
| + processPayment() |
+---------------------+
        ^
        |
+---------------------------+
|   CreditCardPayment       | <--- Implements Payment
|---------------------------|
| + processPayment()        |
+---------------------------+
```

---

b. Java Class Implementing Multiple Interfaces

Real-World Example: Multifunction Printer

Consider a device that can perform both printing and scanning operations. The device can be represented by a class that implements two interfaces: Printable and Scannable.

Interface Definitions

java
CopyEdit
```
interface Printable {
    void printDocument(String document);
}
```

```
interface Scannable {
    void scanDocument(String document);
}
```

Class Implementing Both Interfaces

```
java
CopyEdit
class MultiFunctionPrinter implements Printable, Scannable {
    public void printDocument(String document) {
        System.out.println("Printing document: " + document);
    }

    public void scanDocument(String document) {
        System.out.println("Scanning document: " + document);
    }
}
```

4. What is Inheritance in Object-Oriented Paradigm? Explain Various Types of Inheritance with Diagram and Example.

Inheritance in Object-Oriented Programming

Inheritance is a key concept in object-oriented programming (OOP) that enables a class, known as a **subclass (child class)**, to inherit attributes and methods from another class, called the **superclass (parent class)**. This feature enhances **code reusability**, **modularity**, and promotes a **hierarchical structure** of classes. In Java, inheritance is implemented using the **extends** keyword

1. Single Inheritance

A subclass inherits from a single superclass. This is the most basic form of inheritance.

Diagram:

```
diff
CopyEdit
+-------------+
| Superclass  |
+-------------+
      |
      v
+-------------+
|  Subclass   |
+-------------+
```

Example:

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

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

```
}
```

2. Multilevel Inheritance

This occurs when a subclass derives from another subclass, forming a chain of inheritance.

Diagram:

```
diff
CopyEdit
+-------------+
| Superclass  |
+-------------+
      |
      v
+-------------+
| Subclass1   |
+-------------+
      |
      v
+-------------+
| Subclass2   |
+-------------+
```

Example:

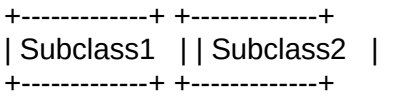
```
java
CopyEdit
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

class Mammal extends Animal {
    void walk() {
        System.out.println("This mammal walks.");
    }
}

class Dog extends Mammal {
    void bark() {
        System.out.println("The dog barks.");
    }
}
```

3. Hierarchical Inheritance

In this type, multiple subclasses inherit from a single superclass.

Diagram:

```
lua
CopyEdit
      +-------------+
      | Superclass  |
      +-------------+
      /          \
     v            v
```



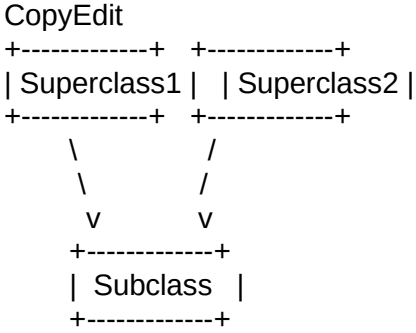
Example:

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

class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("The cat meows.");
    }
}
```

4. Multiple Inheritance (via Interfaces)

Java does **not** support multiple inheritance through classes to prevent **ambiguity** and **complexity**. However, it can be implemented using **interfaces**.

Diagram:



Example using Interfaces:

```
java
CopyEdit
interface Canine {
    void bark();
}

interface Pet {
    void play();
}

class Dog implements Canine, Pet {
    public void bark() {
```

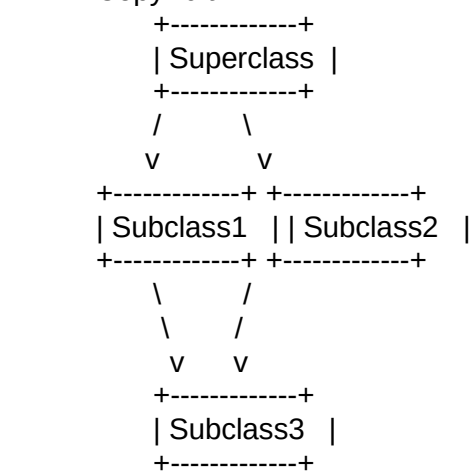
```
        System.out.println("The dog barks.");
    }

    public void play() {
        System.out.println("The dog plays.");
    }
}
```

5. Hybrid Inheritance (via Interfaces)

Hybrid inheritance is a combination of **multiple** and **hierarchical inheritance**. Java supports hybrid inheritance using **interfaces**.

Diagram:

lua
CopyEdit



Example using Interfaces:

java
CopyEdit

```
interface Animal {
    void eat();
}

interface Mammal extends Animal {
    void walk();
}

class Dog implements Mammal {
    public void eat() {
        System.out.println("The dog eats.");
    }

    public void walk() {
        System.out.println("The dog walks.");
    }
}
```

5. What are Superclass, Subclass, and Polymorphism Concepts in Inheritance? Explain with the Help of Diagram and Java Syntax.

Superclass and Subclass in Java

- **Superclass**: The class from which attributes and methods are inherited. Also referred to as the **parent class** or **base class**.
- **Subclass**: The class that derives attributes and behaviors from a superclass. Also known as the **child class** or **derived class**.

Diagram:

diff
CopyEdit
```
+-------------+
| Superclass  |
+-------------+
      ^
      |
+-------------+
|  Subclass   |
+-------------+
```

Java Syntax:

java
CopyEdit
```
class Superclass {
    // Attributes and methods
}

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

Polymorphism in Java

Polymorphism enables an object to be treated as an instance of its **superclass** rather than its actual derived class. This allows a single interface to handle multiple data types efficiently.

Example:

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

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

class Cat extends Animal {
    void makeSound() {
        System.out.println("Cat meows");
    }
}

public class TestPolymorphism {
    public static void main(String[] args) {
        Animal a; // Superclass reference
```

```
        a = new Dog(); // Upcasting
        a.makeSound(); // Calls Dog's makeSound()

        a = new Cat(); // Upcasting
        a.makeSound(); // Calls Cat's makeSound()
    }
}
```

Mini Project Idea: Library Management System (LMS)

Project Overview

The **Library Management System (LMS)** is designed to oversee the library's operations, including **book tracking, user management, borrowing, and returning processes**. The system ensures smooth management of book inventories and user interactions.

Classes, Attributes, and Methods

1. Book Class

Attributes:

- bookID: Unique identifier for each book.
- title: Name of the book.
- author: Author's name.
- publisher: Publisher of the book.
- isAvailable: Boolean indicating if the book is available for borrowing.

Methods:

- getDetails(): Retrieves and displays book information.
- checkAvailability(): Checks if the book is available for borrowing.

2. User Class

Attributes:

- userID: Unique identifier for each user.
- name: Full name of the user.
- email: Contact email of the user.
- phone: Contact number.

Methods:

- register(): Registers a new user in the system.
- updateProfile(): Updates user information.

3. Member Class (Inherits from User)

Attributes:

- membershipID: Unique membership number.
- borrowedBooks: List of currently borrowed books.

Methods:

- borrowBook(Book book): Allows a member to borrow a book.

- returnBook(Book book): Enables a member to return a borrowed book.

4. Librarian Class (Inherits from User)

Attributes:

- employeeID: Unique identification number for a librarian.

Methods:

- addBook(Book book): Adds a book to the library inventory.
- removeBook(Book book): Deletes a book from the system.
- manageUsers(): Manages user registrations and profile

5. Transaction Class

Attributes:

- transactionID: Unique identifier for each transaction.
- book: The book being borrowed or returned.
- member: The member involved in the transaction.
- issueDate: Date when the book was issued.
- dueDate: Expected return date.
- returnDate: Actual return date.

Methods:

- createTransaction(): Logs a new borrowing transaction.
- closeTransaction(): Marks a transaction as completed upon return.

Inheritance Implementation

In this system:

- The **Member** and **Librarian** classes inherit from the **User** class.
- Common attributes such as userID, name, email, and phone are **defined in the User class** and inherited by both Member and Librarian.
- This structure **reduces code duplication** and provides **a clear class hierarchy**, improving **code reusability and maintainability**.

7. Java Application Design for Specific Domains

Education Domain: Education Management System

The **Education Management System** is designed to handle multiple entities such as **teachers, students, administrative staff, and the accounts department**. By utilizing **inheritance and interfaces**, we can structure the system to be **efficient, modular, and scalable**.

Classes, Attributes, and Methods

1. Superclass: Person (Base Class)

Represents a common entity with shared attributes across all individuals in the system.

Attributes:

- personID: Unique identifier for each person.

- name: Full name.
- address: Residential address.
- phone: Contact number.

Methods:

- getDetails(): Retrieves personal details.

2. Subclass: Teacher (Inherits from Person)

Represents an instructor responsible for teaching and student assessment.

Attributes:

- employeeID: Unique identification number for teachers.
- subjects: List of subjects the teacher teaches.

Methods:

- assignGrades(Student student): Assigns grades to a student.
- prepareLesson(): Prepares a lesson plan.

3. Subclass: Student (Inherits from Person)

Represents a learner enrolled in various courses.

Attributes:

- studentID: Unique student identification number.
- courses: List of courses the student is enrolled in.

Methods:

- enrollCourse(Course course): Enrolls the student in a new course.
- viewGrades(): Displays the student's grades.

4. Subclass: AdministrativeStaff (Inherits from Person)

Represents staff responsible for managing institutional operations.

Attributes:

- staffID: Unique staff identifier.
- department: Department to which the staff member belongs.

Methods:

- manageRecords(): Handles institutional records.

5. Independent Class: AccountDepartment

Handles financial aspects, including budgeting and salary processing.

Attributes:

- budget: Total allocated budget.
- expenses: List of expenses.

Methods:

- processSalary(Teacher teacher): Processes salary payments for teachers.
- manageBudget(): Oversees budget allocation.

Interfaces and Inheritance

To enforce common behavior across multiple roles, **interfaces** are implemented.

Interface: Evaluatable

Defines an evaluation structure for teachers.

Methods:

- evaluateStudent(Student student): Evaluates a student's academic performance.

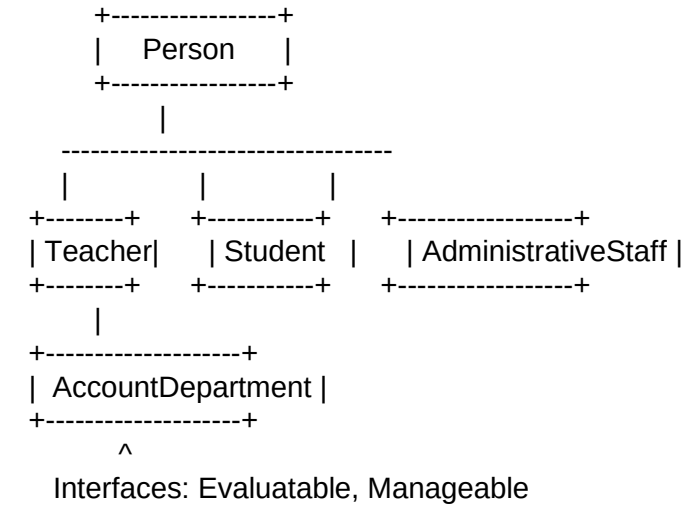
Interface: Manageable

Defines methods for handling records and generating reports.

Methods:

- manageRecords(): Manages institutional records for students, teachers, and staff.
- generateReports(): Generates educational reports.

Class Hierarchy Diagram



Java Code Implementation

```
// Superclass: Person
class Person {
    String personID, name, address, phone;

    void getDetails() {
        System.out.println("Name: " + name + ", Address: " + address);
    }
}

// Interface for student evaluation
interface Evaluatable {
    void evaluateStudent(Student student);
```

```
}

// Subclass: Teacher (inherits Person, implements Evaluatable)
class Teacher extends Person implements Evaluatable {
    String employeeID;
    String subjects;

    public void evaluateStudent(Student student) {
        System.out.println("Evaluating student: " + student.name);
    }

    void assignGrades(Student student) {
        System.out.println("Grades assigned to: " + student.name);
    }

    void prepareLesson() {
        System.out.println("Lesson prepared.");
    }
}

// Subclass: Student (inherits Person)
class Student extends Person {
    String studentID;
    String courses;

    void enrollCourse(String course) {
        System.out.println("Enrolled in: " + course);
    }

    void viewGrades() {
        System.out.println("Viewing grades...");
    }
}

// Subclass: AdministrativeStaff (inherits Person)
class AdministrativeStaff extends Person {
    String staffID, department;

    void manageRecords() {
        System.out.println("Managing records...");
    }
}

// Independent Class: AccountDepartment
class AccountDepartment {
    double budget;

    void processSalary(Teacher teacher) {
        System.out.println("Salary processed for: " + teacher.name);
    }

    void manageBudget() {
        System.out.println("Managing budget...");
    }
}

// Main Class
public class EducationSystem {
    public static void main(String[] args) {
        Teacher t1 = new Teacher();
```

```
        t1.name = "John Doe";

        Student s1 = new Student();
        s1.name = "Alice";

        t1.evaluateStudent(s1);
        t1.assignGrades(s1);
    }
}
```

8. What is a Java Package? What is the Use of Java Packages? What Directory Structure is Followed When We Create a Package? What Naming Conventions are Used When We Write Java Packages?

Java Packages: An Overview

A **package** in Java serves as a **namespace** that organizes a collection of related **classes and interfaces**. It functions similarly to a folder in a file system, where related files are stored together. Packages play a crucial role in **structuring code**, **preventing naming conflicts**, and **managing access control**.

Benefits of Using Java Packages

1. **Efficient Organization**: Packages help in structuring the codebase by grouping related classes and interfaces, making it easier to manage and navigate.
2. **Avoiding Naming Conflicts**: When classes are placed in different packages, multiple classes with the same name can coexist without conflicts.
3. **Access Control**: Packages enable access protection, allowing **package-private members** to be accessed only within the same package.
4. **Code Reusability**: Developers can reuse pre-written classes and interfaces across multiple projects by grouping them into packages.

Directory Structure for Java Packages

In Java, the directory structure of a package corresponds to its **namespace**. For instance, if a package is named **com.example.project**, it will be structured as follows:

markdown
CopyEdit
```
com/
└── example/
    └── project/
```

Each **dot (.)** in the package name represents a new **subdirectory level** in the file system

Java Package Naming Conventions

To ensure consistency, Java follows specific conventions for naming packages:

- **Lowercase Letters**: Package names should be entirely in **lowercase** to distinguish them from class or interface names.
- **Reverse Domain Name**: Organizations typically use their **reversed domain name** as the base package name. For example, a company with the domain example.com would use com.example as the root package.
- **No Underscores**: When a package name consists of multiple words, they are concatenated without underscores. Example: formvalidator instead of form_validator.

Built-in Java Packages

Java provides several **predefined packages** as part of the **Java Development Kit (JDK)**. These built-in packages contain **ready-to-use** classes and interfaces that facilitate various programming tasks, such as **data structures, networking, and I/O operations**.

Examples of Built-in Packages

1. **java.lang**: Contains fundamental classes such as String, Math, Integer, and System. This package is **automatically imported** into every Java program.
2. **java.util**: Includes utility classes for handling **collections** (List, Set, Map), **date and time operations**, and **random number generation**.
3. **java.io**: Provides classes for **input and output operations**, including File, InputStream, OutputStream, Reader, and Writer.
4. **java.net**: Contains networking-related classes like Socket, ServerSocket, and URL, enabling the development of network applications.
5. **java.awt**: Includes classes for building **graphical user interfaces (GUIs)**, handling **graphics**, and rendering **images**.
6. **javax.swing**: Provides **lightweight** GUI components that work consistently across platforms.
7. **java.sql**: Offers classes and interfaces for working with **databases**, enabling applications to **store, retrieve, and manage data** efficiently.

10. What is a User-Defined Package in Java? Explain with an Example. Which Commands Will You Use to Create a User-Defined Package? What is a User-Defined Package?

User-Defined Packages in Java

A **user-defined package** in Java is a package created by a programmer to **organize** and **group related classes, interfaces, and sub-packages**. This helps maintain a structured project, **prevents naming conflicts**, and **enhances reusability**.

In Java, the package keyword is used to define a user-defined package.

Steps to Create a User-Defined Package

1. **Defining the Package:**
   - Use the package keyword at the beginning of the Java file.
2. **Compiling the Java File:**
   - Use the command:

     nginx
     CopyEdit
     javac -d . ClassName.java

     This ensures the compiled file is stored in the appropriate package directory.

3. **Importing and Using the Package:**
   - Use the import statement to access classes from the package in another Java program.

Example of Creating and Using a User-Defined Package

Step 1: Create the Package

Save the following file as MyPackageClass.java inside a folder named mypackage.

java

CopyEdit
```
// Define package
package mypackage;

public class MyPackageClass {
    public void displayMessage() {
        System.out.println("Hello from MyPackageClass!");
    }
}
```

Step 2: Compile the Java File

Run the following command in the terminal or command prompt:

sh
CopyEdit
```
javac -d . MyPackageClass.java
```

This command creates a directory named mypackage in the current location, and the compiled .class file is placed inside it.

Step 3: Use the Package in Another Class

Create another Java file TestPackage.java to import and use the user-defined package.

java
CopyEdit
```
// Import the user-defined package
import mypackage.MyPackageClass;

public class TestPackage {
    public static void main(String[] args) {
        MyPackageClass obj = new MyPackageClass();
        obj.displayMessage(); // Accessing method from the package
    }
}
```

Step 4: Compile and Run the Program

Compile the TestPackage.java file:

sh
CopyEdit
```
javac TestPackage.java
```

Run the program:

sh
CopyEdit
```
java TestPackage
```

Output:

csharp
CopyEdit
```
Hello from MyPackageClass!
```

Java Commands for Creating and Using a User-Defined Package

Step	Command
Compile the package	javac -d . MyPackageClass.java
Compile the main class	javac TestPackage.java
Run the program	java TestPackage

Advantages of User-Defined Packages in Java

1. **Better Code Organization** – Groups related classes for a well-structured codebase.
2. **Encapsulation** – Enhances security by controlling access to package members.
3. **Prevents Naming Conflicts** – Avoids duplicate class names in different projects.
4. **Reusability** – Once created, packages can be reused in various applications.

11. Proposed Package Structure for the Library Management System Mini Project

Organizing a Package Structure for a Library Management System (LMS)

Designing a well-structured package hierarchy is essential for maintaining and scaling a **Java project**. A **feature-based package organization** is ideal for a **Library Management System (LMS)** as it groups related classes based on their functionality, ensuring **better modularity and readability**.

Proposed Package Structure

1. com.librarymanagement

- **model** (Represents core entities)
  - **Classes:**
    - Book
    - User
    - Member
    - Librarian
    - Transaction
- **service** (Contains business logic and operations)
  - **Classes:**
    - BookService
    - UserService
    - TransactionService
- **dao** (Data Access Object - Manages database interactions)
  - **Classes:**
    - BookDAO
    - UserDAO
    - TransactionDAO
- **ui** (Handles user interface components)
  - **Classes:**
    - MainMenu
    - BookUI
    - UserUI
    - TransactionUI
- **util** (Contains utility functions and helper classes)
  - **Classes:**
    - DatabaseConnection
    - DateUtil

Explanation of Each Package

1. model Package

- Defines the core data structures and entities used in the **LMS**.
- Example:
  - The Book class might include attributes such as bookID, title, author, and methods like getDetails() and checkAvailability().

2. service Package

- Implements the **business logic** and core functionalities of the system.
- Example:
  - The BookService class may contain methods like addBook(), removeBook(), and searchBooks().

3. dao Package

- Manages **database interactions**, ensuring correct data storage and retrieval.
- Example:
  - The BookDAO class would handle SQL operations related to the **Book** entity, such as fetchBookById() or saveBook().

4. ui Package

- Manages **user interactions** by presenting the interface to users.
- Example:
  - The MainMenu class could display options like **"Add Book"**, **"Issue Book"**, and **"Return Book"**.

5. util Package

- Provides **utility classes and helper functions** used across the system.
- Example:
  - DatabaseConnection manages database connectivity.
  - DateUtil offers date formatting and manipulation utilities.