Program -1

Coffee Café Order System Program Definition:

This Java program simulates a basic coffee café ordering system using interfaces, classes, and packages. The program allows users to create multiple coffee orders, select coffee types and sizes, and calculates the total price based on their choices. Here's an explanation of the key components:

1. Package and Interface (`CoffeeOrder`):

- The `com.cafe` package encapsulates all the classes related to the café system.
- **Interface (`CoffeeOrder`)**: Defines the contract for any class that implements it. It declares the methods required for a coffee order, such as:
  - `selectCoffee(String coffeeType)`: Allows the user to choose the type of coffee (e.g., Espresso, Latte, Cappuccino).
  - `selectSize(String size)`: Allows the user to choose the size of the coffee (Small, Medium, Large).
  - `displayOrder()`: Displays the details of the order (coffee type, size, and price).
  - `getPrice()`: Returns the price of the current coffee order.

2. Implementation Class (`CafeOrder`):

- The `CafeOrder` class implements the `CoffeeOrder` interface and provides the functionality for selecting coffee types and sizes, calculating prices, and displaying the order.
- **Methods**:
  - **`selectCoffee(String coffeeType)`**: Sets the type of coffee selected by the user. Based on the coffee type, it assigns a base price (e.g., Espresso is $3, Latte is $4, etc.).
  - **`selectSize(String size)`**: Modifies the price based on the size chosen (e.g., Medium adds $1, Large adds $2).
  - **`displayOrder()`**: Displays the current order, including the selected coffee type, size, and total price.
  - **`getPrice()`**: Returns the calculated price for the order.

3. Main Class (`Main`):

- The `Main` class is the entry point of the program, responsible for handling user interaction, allowing multiple coffee orders, and calculating the total price.
- **Functionality**:
  - **Multiple Orders**: The program uses a loop to allow users to create multiple coffee orders. After each order, the user is prompted to either continue ordering or stop.
  - **Order List**: Each coffee order is stored in an `ArrayList` called `orders`. This allows the program to keep track of all the orders made by the user.
  - **Order Summary and Total Price**: After all orders are created, the program iterates through the `orders` list, displaying each order's details and calculating the total price for all the orders.

Program -2

Torrent Bill Payment Kiosk System Definition

The **Torrent Bill Payment Kiosk System** is a Java-based application designed to facilitate the payment of electricity bills through a self-service kiosk. This system allows customers to pay their bills without waiting in long queues, providing a more efficient and user-friendly experience. The application employs the **Chain of Responsibility** design pattern, allowing for modular handling of different currency denominations during the payment process.

Key Components of the System:

1. **Package Structure**:
   - The program is organized into a package named `com.kiosk`, which encapsulates all related classes and interfaces. This modular design promotes better code organization and maintainability.
2. **Handler Interface (`DenominationHandler`)**:
   - An interface that defines the methods for handling requests related to currency denominations. It includes methods for setting the next handler in the chain and processing the payment request.
3. **Concrete Handlers**:
   - **`FiveHundredHandler`**: Responsible for processing requests for 500 rupee notes. It checks if the amount is sufficient, calculates the number of notes to fetch, and passes the remaining amount to the next handler.
   - **`OneHundredHandler`**: Handles requests for 100 rupee notes in a similar manner, processing any remaining amount after handling 500 rupee notes.
   - **`TenHandler`**: Responsible for 10 rupee notes, fetching as many as needed based on the remaining amount.
   - **`FiveHandler`**: Processes requests for 5 rupee notes, handling any remaining amount after the previous handlers.
4. **Kiosk Class (`Kiosk`)**:
   - This class manages the payment process. It creates a chain of handlers for different currency denominations, setting them up in the order of 500, 100, 10, and 5 rupees. The `payBill` method is invoked to initiate the payment process, passing the specified amount to the first handler in the chain.
5. **Main Class**:
   - The entry point of the application, where the user interacts with the system. It prompts the customer to enter their customer number and the amount to be paid. The entered amount is then processed by the `Kiosk` class, which utilizes the chain of handlers to dispense the correct notes.

Program -3

Definition of Banking System in Java

Overview

The **Banking System** is a simple console-based application that allows users to manage their bank accounts through operations such as creating accounts, depositing money, withdrawing funds, and checking account balances. The application is designed using object-oriented principles, with custom exception handling to manage errors gracefully.

Components

1. **Packages**:
   - `com.bank`: Contains core banking functionalities.
   - `com.bank.exceptions`: Contains custom exceptions related to banking operations.
2. **Classes**:
   - **BankingException**: Custom exception class for handling banking-related errors.
   - **BankingOperations**: Interface defining the methods for banking operations (deposit, withdraw, get balance).
   - **Account**: Implements the `BankingOperations` interface. Represents a bank account and manages balance, deposits, and withdrawals.
   - **BankingSystem**: Manages multiple accounts and provides methods to create and retrieve accounts.
   - **Main**: Entry point for user interaction, providing a command-line interface for users to perform banking operations.

Functionalities

1. **Create Account**:
   - Allows users to create a new bank account by entering a unique account number.
   - Checks if the account already exists and ensures the maximum number of accounts is not exceeded.
2. **Deposit Money**:
   - Users can deposit a specified amount into their account.
   - Validates the deposit amount (must be greater than zero) and updates the account balance.
   - Throws a `BankingException` if the deposit amount is invalid.
3. **Withdraw Money**:
   - Users can withdraw a specified amount from their account.
   - Validates the withdrawal amount (must be greater than zero and not exceed the current balance).
   - Updates the account balance upon successful withdrawal.
   - Throws a `BankingException` if the withdrawal amount is invalid or if there are insufficient funds.
4. **Check Balance**:
   - Allows users to view their current account balance.
   - Does not involve exception handling as retrieving the balance is a straightforward operation.

5. **User Interaction**:
    - The `Main` class provides a simple command-line interface to interact with the user.
    - Users can choose different operations by entering corresponding options, with clear prompts and error messages for invalid inputs.

Exception Handling

- The system uses custom exceptions (`BankingException`) to manage errors related to banking operations.
- `try-catch` blocks are employed in the `deposit` and `withdraw` methods to handle potential errors gracefully, informing the user of any issues without crashing the application.

Program -4

hospital/
│
├── Doctor.java
├── Specialist.java
├── Appointment.java
└── Main.java

Doctor Appointment System

Objective:

Develop a Java program that simulates a doctor appointment system using the concepts of **abstraction**, **inheritance**, and **polymorphism**. The program should demonstrate how different types of doctors handle appointment bookings through a structured class hierarchy.

Requirements:

1. **Create an Abstract Class**:
   - Define an abstract class named `Doctor` in the package `hospital`.
   - Include private attributes for the doctor's `name` and `specialization`.
   - Provide a constructor to initialize these attributes.
   - Implement public getter methods for these attributes.
   - Declare an abstract method named `bookAppointment()`.
2. **Create a Subclass**:
   - Create a class named `Specialist` in the package `hospital.doctors` that extends the `Doctor` class.
   - Add an additional private attribute for the doctor's `expertise`.
   - Implement a constructor to initialize `name`, `specialization`, and `expertise`.
   - Override the `bookAppointment()` method to provide a specific implementation that displays a message about booking an appointment with the specialist.
3. **Create an Appointment Class**:
   - Define a class named `Appointment` in the `hospital` package.
   - Include a private attribute to hold a reference to a `Doctor` object (which can be a `Doctor` or `Specialist`).
   - Provide a constructor to initialize the `Doctor` attribute.
   - Implement a method named `schedule()` that calls the `bookAppointment()` method on the `Doctor` reference.
4. **Implement a Main Class**:
   - Create a class named `Main` to serve as the entry point of the program.
   - In the `main` method:
     - Instantiate a `Specialist` object.
     - Create an `Appointment` object using the `Specialist`.
     - Call the `schedule()` method to demonstrate booking an appointment.

Program -5

Banking System Simulation

Objective:

Develop a Java program that simulates a basic banking system, allowing users to perform essential banking operations, including depositing money, withdrawing money, checking their balance, and viewing a mini statement of transactions.

Requirements:

1. **Create a Class for Bank Account**:
    - Define a class named `BankAccount` that represents a bank account.
    - Include the following attributes:
        - `accountHolder`: a `String` to store the name of the account holder.
        - `balance`: a `double` to hold the current balance of the account.
        - `transactions`: a `List<String>` to store a history of transactions.
    - Implement the following methods:
        - **Constructor**: To initialize the account holder's name and set the balance to 0.0.
        - `deposit(double amount)`: To add the specified amount to the balance and record the transaction. Ensure that the deposit amount is positive.
        - `withdraw(double amount)`: To subtract the specified amount from the balance if sufficient funds are available. Record the transaction. Ensure that the amount is valid.
        - `printMiniStatement()`: To display a summary of transactions and the current balance.
2. **Create the Banking System Class**:
    - Define a class named `BankingSystem` with a `main` method that serves as the entry point of the program.
    - Use the `Scanner` class to handle user input.
    - Implement a menu-driven interface using a `do-while` loop that displays the following options:
        - Deposit
        - Withdraw
        - Check Balance
        - Mini Statement
        - Exit
    - Use a switch-case structure to handle user choices and call the appropriate methods of the `BankAccount` class based on the user's selection.
3. **User Interaction**:
    - Prompt the user to enter their name when the program starts and create a `BankAccount` instance using that name.
    - Ensure that the program continues to run until the user chooses to exit.

Provide appropriate messages for each operation and handle invalid inputs gracefully

Program -6

Property Management System

Create a property interface that has two basic functionalities – Buy and sell. Every property item can be bought as well as sold. Create three different classes – Apartments, Bungalow and Tenaments. These three different classes can implement the property class created as a component interface. The above threeclasses are called as leaf classes as they implement a component interface. Finally, create a class calledApplication that uses a collection of different types of property.

- **Create a Package**:
  - Create a package named `propertymanagement` to contain all the classes.
- **Create a Property Interface**:
  - Define an interface named `Property` in a file named `Property.java` within the `propertymanagement` package that declares the following methods:
    - `void buy()`: A method to represent buying a property.
    - `void sell()`: A method to represent selling a property.
- **Create Property Classes**:
  - **Apartments Class**:
    - Create a class named `Apartments` in a file named `Apartments.java` that implements the `Property` interface.
    - Include the following attributes:
      - `location`: a `String` to store the apartment's location.
      - `price`: a `double` to store the price of the apartment.
    - Implement the methods from the `Property` interface to print messages when buying and selling an apartment.
  - **Bungalow Class**:
    - Create a class named `Bungalow` in a file named `Bungalow.java` that implements the `Property` interface.
    - Include attributes for `location` and `price`.
    - Implement the methods to print messages for buying and selling a bungalow.
  - **Tenaments Class**:
    - Create a class named `Tenaments` in a file named `Tenaments.java` that implements the `Property` interface.
    - Include attributes for `location` and `price`.
    - Implement the methods to print messages for buying and selling a tenament.
- **Create an Application Class**:
  - Define a class named `Application` in a file named `Application.java` with a `main` method that serves as the entry point of the program.
  - In the `main` method:
    - Create a list (e.g., `ArrayList`) to hold different types of properties.

- Instantiate at least one object of each property class (`Apartments`, `Bungalow`, and `Tenaments`) and add them to the list.
- Use a loop to iterate through the list and call the `buy()` and `sell()` methods for each property, demonstrating polymorphism.

Program -7

Flour Management System:-
Create Consider a class Flour with two attributes – weight and price. Create a class FlourItem with a
method to get the object of Flour class with default values. Create an Interface FlourItemInterface that contains 3 different methods to provide flour packets of 3 different sizes (for example: public Flour getQuintal(); public Flour get10kg();public Flour get1kg();). Implement the interface and set the relevant pricing.

1. **Create a Flour Class**:
    - Define a class named `Flour` with the following attributes:
        - `weight` (double): Represents the weight of the flour in kilograms.
        - `price` (double): Represents the price per kilogram of the flour.
    - Include a constructor to initialize these attributes and appropriate getter methods.
    - Implement a method to calculate the total price of the flour based on its weight.
2. **Create a FlourItem Class**:
    - Define a class named `FlourItem` that includes a method to return a `Flour` object with default values (e.g., 1 kg of flour priced at a specified rate).
3. **Create a FlourItemInterface**:
    - Define an interface named `FlourItemInterface` that declares the following methods:
        - `Flour getQuintal()`: Method to return a `Flour` object representing a 100 kg flour packet.
        - `Flour get10kg()`: Method to return a `Flour` object representing a 10 kg flour packet.
        - `Flour get1kg()`: Method to return a `Flour` object representing a 1 kg flour packet.
4. **Create a FlourStore Class**:
    - Implement a class named `FlourStore` that implements the `FlourItemInterface`.
    - In this class, provide the actual implementation of the methods to create and return `Flour` objects with appropriate weights and prices for each packet size:
        - For example, the price for the 100 kg packet should be lower per kilogram compared to smaller packets.
5. **Create an Application Class**:
    - Define a class named `Application` with a `main` method that serves as the entry point of the program.
    - In the `main` method:
        - Create an instance of `FlourStore`.
        - Use the methods from the `FlourItemInterface` to obtain flour packets of different sizes.
        - Display the weight and total price of each flour packet using the methods from the `Flour` class.

Program -8

Library Management System

Develop a Java program that simulates a Library Management System. The system will allow users to add books, lend books to members, and return books, utilizing abstraction and inheritance through a package structure.

Requirements:

1. **Create a Package**:
   - Organize all your classes in a package named `library`.
2. **Define an Abstract Class**:
   - Create an abstract class named `AbstractBook` with the following attributes:
     - `String title`: Represents the title of the book.
     - `String author`: Represents the author of the book.
     - `boolean isLent`: Represents the lending status of the book (true if lent, false if available).
   - Include:
     - A constructor to initialize the attributes.
     - Getter methods for the title and author.
     - Methods to lend and return the book.
     - An abstract method named `getBookType()` that must be implemented by subclasses.
     - A `toString()` method to display the book's details.
3. **Create a Book Class**:
   - Implement a concrete class named `Book` that extends `AbstractBook`.
   - Implement the `getBookType()` method to return a string indicating the book type (e.g., "Regular Book").
4. **Create a Library Class**:
   - Implement a class named `Library` to manage a collection of `AbstractBook` objects. It should include:
     - A list to hold books.
     - Methods to:
       - Add a book to the library.
       - Lend a book based on its title.
       - Return a book based on its title.
       - List all books in the library.
5. **Create an Application Class**:
   - Implement a class named `Application` with a `main` method to serve as the entry point of the program. The application should:
     - Provide a menu-driven interface to allow users to:
       - Add books (prompting for title and author).
       - Lend books (prompting for the book title).
       - Return books (prompting for the book title).
       - List all available books.
       - Exit the application.
     - Use a `Scanner` to handle user input.

Program -9

Java Applets (pending)

Program -10

Java Applets (pending)