1) Assignment Definition: Car Rental System

Program Description:

You are tasked with creating a Java program for a car rental system. The program should model different types of vehicles using the `Vehicle`, `Car`, and `SportsCar` classes.

Requirements:

1. **Vehicle Class:**
   - Create a `Vehicle` class with the following properties:
     - `name` (String): The name of the vehicle.
     - `manufacturer` (String): The manufacturer of the vehicle.
   - Implement a method `startEngine()` that prints a message indicating that the vehicle's engine is started.
2. **Car Class:**
   - Create a `Car` class that extends `Vehicle` with the following additional property:
     - `numDoors` (int): The number of doors in the car.
   - Implement a method `honkHorn()` to simulate the car honking its horn.
3. **SportsCar Class:**
   - Create a `SportsCar` class that extends `Car` with the following additional property:
     - `topSpeed` (int): The top speed of the sports car.
   - Implement a method `activateTurbo()` to simulate activating the turbo mode of the sports car.
4. **Main Program:**
   - In the main program, create instances of `Vehicle`, `Car`, and `SportsCar` to represent different types of vehicles.
   - Demonstrate the use of methods and properties of these classes by starting the engine, honking the horn, and activating the turbo (for sports cars).

- Display relevant information about each vehicle, such as its name, manufacturer, number of doors (for cars), and top speed (for sports cars).

---

2) Assignment Definition :-

Abstract `Thali` Class:

- **Definition**: The `Thali` class is an abstract class that serves as the base class for different types of Thalis.
- **Purpose**: It defines the common structure and behavior that all Thali types must follow. It declares abstract methods for adding specific components to a Thali, such as Sabji, Dal, Rice, and Roti, and abstract methods for setting their prices. It also includes a method `makeThali` to print a generic message.
- **Attributes**: It has a private `price` attribute to keep track of the total price of the Thali. This attribute is initialized to 0.0 in the constructor.
- **Methods**:
  - `abstract void addSabji(double price)`: An abstract method for adding Sabji to the Thali. It takes a `price` parameter to set the price of Sabji.
  - `abstract void addDal(double price)`: An abstract method for adding Dal to the Thali. It takes a `price` parameter to set the price of Dal.
  - `abstract void addRice(double price)`: An abstract method for adding Rice to the Thali. It takes a `price` parameter to set the price of Rice.
  - `abstract void addRoti(double price)`: An abstract method for adding Roti to the Thali. It takes a `price` parameter to set the price of Roti.
  - `void makeThali()`: A non-abstract method that prints a message indicating that the Veg Thali will be ready in 30 minutes.
  - `public double getPrice()`: A method to retrieve the total price of the Thali.
  - `protected void setPrice(double price)`: A method to accumulate the price of components added to the Thali.

2. **Concrete `GujaratiThali` and `PunjabiThali` Classes**:

- **Definition**: These classes are concrete implementations of the `Thali` class, representing specific types of Thalis (Gujarati and Punjabi).
- **Purpose**: They provide concrete implementations for the abstract methods declared in the `Thali` class, specifying how to add components (e.g., Sabji, Dal, Rice, Roti) and set their prices for each type of Thali.
- **Methods**: Each class overrides the abstract methods (`addSabji`, `addDal`, `addRice`, `addRoti`) and uses the `setPrice` method to accumulate the total price as components are added.

3. `FactoryDesignPattern` **Class**:
    - **Definition**: This class serves as the entry point and demonstration of the Factory design pattern.
    - **Purpose**: It creates instances of different Thalis (Gujarati and Punjabi) using a factory method and demonstrates how to add components with prices and retrieve the total price.
    - **Methods**: The `main` method in this class performs the following actions:
        - Creates an instance of `ThaliRestaurant`, which extends `BaseThaliRestaurant`.
        - Uses the factory method `createThali` to create specific Thalis (Gujarati and Punjabi).
        - Adds components to each Thali with specified prices.
        - Calls the `makeThali` method to indicate the Thali is being prepared.
        - Retrieves and displays the total price of each Thali using the `getPrice` method.

In summary, this program demonstrates the Factory design pattern by creating different types of Thalis (Gujarati and Punjabi) with customizable components and prices. It encapsulates the creation logic within the `ThaliRestaurant` class, providing a consistent interface for creating Thalis while allowing for flexibility and customization of each Thali type.

---

3) Assignment Definition

`PizzaOrderSystem` interface:

1. `PizzaOrderSystem` **Interface**:
    - **Definition**: The `PizzaOrderSystem` interface defines a set of methods that represent operations related to a pizza order system.
    - **Methods**:
        - `placeOrder(String pizzaType, int quantity)`: This method allows placing a new pizza order by specifying the type of pizza (`pizzaType`) and the quantity desired (`quantity`).
        - `checkOrderStatus(int orderId)`: It checks the status of an existing order by providing the order ID (`orderId`) and returns a status message as a `String`.
        - `cancelOrder(int orderId)`: This method cancels an existing order based on the order ID (`orderId`) and returns `true` if the order was successfully canceled or `false` if it couldn't be canceled.
        - `calculateOrderCost(int orderId)`: It calculates the total cost of an existing order based on the order ID (`orderId`) and returns the cost as a `double`.
        - `listAvailablePizzas()`: This method lists available pizza options to provide customers with a menu of choices.
2. `PizzaOrderProcessor` **Class (Concrete Implementation)**:
    - **Definition**: The `PizzaOrderProcessor` class implements the `PizzaOrderSystem` interface, providing concrete implementations for each of the methods defined in the interface.
    - **Methods**:
        - `placeOrder`: Places a new pizza order with a unique order ID, pizza type, and quantity.
        - `checkOrderStatus`: Simulates checking the status of an order based on the order ID (simplified for demonstration).
        - `cancelOrder`: Simulates canceling an order based on the order ID (simplified for demonstration).
        - `calculateOrderCost`: Calculates the cost of an order based on the order ID (simplified for demonstration).
        - `listAvailablePizzas`: Lists available pizza options.
3. `PizzaOrderSystemExample` **Class (Main Program)**:
    - **Definition**: This is the main program class that demonstrates the usage of the `PizzaOrderSystem` interface and the `PizzaOrderProcessor` implementation.

- **Main Method**: In the `main` method, the following actions are performed:
    - An instance of `PizzaOrderProcessor` is created and assigned to the `PizzaOrderSystem` interface.
    - The available pizza options are listed using `listAvailablePizzas`.
    - A new pizza order for "Margherita Pizza" with a quantity of 2 is placed using `placeOrder`.
    - The status of the order is checked using `checkOrderStatus`.
    - The cost of the order is calculated using `calculateOrderCost`.
    - An attempt to cancel the order is made using `cancelOrder`.

This program demonstrates how the `PizzaOrderSystem` interface defines a contract for pizza ordering operations, and the `PizzaOrderProcessor` class provides the concrete implementation of those operations. The `main` method showcases the usage of these methods within a simplified pizza ordering system.

---

4) Assignment Definition

Program Overview:

This Java program models a simple school system using object-oriented programming concepts such as inheritance, encapsulation, and method overriding. It defines three classes: `Person`, `Student`, and `Teacher`, each with its own attributes, methods, and encapsulation techniques.

1. `Person` **Class**:
   - Represents a basic person with attributes `name` and `age`.
   - Provides getter and setter methods for `name` and `age`.
   - The `setAge` method includes a validation check to ensure that age is non-negative.
   - Includes an `introduce` method to print the person's name and age.
2. `Student` **Class**:
   - Extends the `Person` class, inheriting its attributes and methods.
   - Adds an additional attribute `studentId` specific to students.
   - Provides getter and setter methods for `studentId`.
   - Overrides the `introduce` method to include `studentId`.
   - Defines a unique method `study` to simulate a student studying.
3. `Teacher` **Class**:
   - Also extends the `Person` class and inherits its attributes and methods.
   - Adds an attribute `subject` specific to teachers.
   - Provides getter and setter methods for `subject`.
   - Overrides the `introduce` method to include `subject`.
   - Defines a unique method `teach` to simulate a teacher teaching.
4. `SchoolSystem` **Class**:
   - The `main` method serves as the entry point of the program.
   - It creates instances of a `Student` and a `Teacher`.
   - Demonstrates the use of setter methods to update attributes (e.g., changing a student's age and a teacher's subject).
   - Invokes methods like `introduce`, `study`, and `teach` to showcase polymorphism and method overriding.

Overall, this program demonstrates the principles of inheritance, encapsulation, and method overriding in a school system context, providing a clear separation of concerns and a structured object-oriented design.

---

5) Assignment Definition

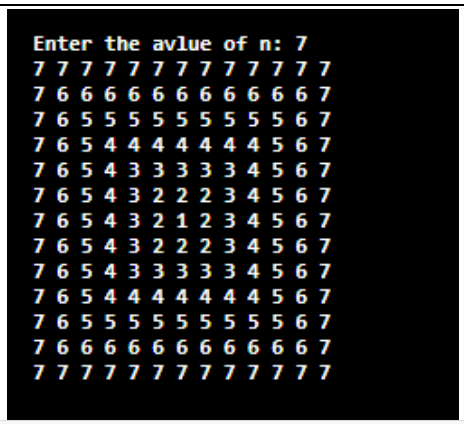
The Java program provided is a simple example of object-oriented programming (OOP) and inheritance in a university department system. Here's an explanation of the key components of the program:

1. **Classes**: The program defines several classes to represent different entities within a university department system.
    - `User` class: This is the base class that represents a generic user with properties such as `username` and `email`. It has getter methods to retrieve these properties.
    - `Professor` class: This class extends the `User` class, inheriting its properties (`username` and `email`) and methods. It adds an additional property, `department`, to represent the department to which the professor belongs. The constructor of this class initializes the properties, and it has a getter method to retrieve the `department`.
    - `Course` class: This class represents a university course with properties like `code`, `name`, and `creditHours`. Similar to the other classes, it has getter methods to access these properties.
    - `Department` class: This class represents a university department. It has properties like `name`, `professor1`, and `course1`. It can be associated with a professor and a course, allowing for the modeling of department-specific information.
2. **Main Method**: The `UniversityDepartmentSystem` class contains the `main` method, which serves as the entry point for the program.
3. **Program Flow**: Inside the `main` method, the program performs the following actions:
    - It creates instances of `Professor` and `Course` to represent a professor and a course within a department.
    - It creates an instance of the `Department` class, specifically the "Computer Science" department (`csDepartment`).
    - The program sets the professor and course for the department using setter methods (`setProfessor1` and `setCourse1`).

- Finally, the program displays information about the department, including the department name, professor's username, email, and department, as well as details about the course, such as its code, name, and credit hours.

This Java program illustrates concepts of inheritance, where the `Professor` class inherits properties and methods from the `User` class, and composition, where the `Department` class can contain instances of other classes (a professor and a course) to model relationships between different entities in a university department system. It provides a basic structure for managing and displaying information related to professors, courses, and departments within a university context.

6) Assignment Definition



This Java program is designed to print a pattern with a given number `n` as input. The pattern consists of four parts:

1. Upper-left part: It prints numbers in ascending order starting from 1.
2. Upper-right part: It also prints numbers in ascending order starting from 1.
3. Lower-left part: It prints numbers in descending order starting from `n`.
4. Lower-right part: It also prints numbers in descending order starting from `n`.

7) Assignment Definition

Find GCD of two numbers using while loop and if else statement

8) Assignment Definition

Java Program to Add Two Matrix Using Multi-dimensional Arrays

9) Assignment Definition

program to check prime number using recursion

10) Assignment Definition

1. `public class StringBufferExample {`: This line defines a Java class named `StringBufferExample`.
2. `public static void main(String[] args) {`: This line starts the `main` method, which is the entry point for the program.
3. `StringBuffer stringBuffer = new StringBuffer("Hello, World!");`: This line creates a `StringBuffer` object named `stringBuffer` and initializes it with the string "Hello, World!".
4. `stringBuffer.append(" Welcome to Java!");`: This line appends the string " Welcome to Java!" to the `stringBuffer`.
5. `stringBuffer.insert(12, "from ");`: This line inserts the string "from " at position 12 in the `stringBuffer`.
6. `stringBuffer.replace(7, 12, "Universe");`: This line replaces characters from position 7 to 11 with the string "Universe" in the `stringBuffer`.
7. `stringBuffer.setCharAt(0, 'h');`: This line sets the character at position 0 in the `stringBuffer` to 'h'.
8. `stringBuffer.delete(2, 5);`: This line deletes characters from position 2 to 4 in the `stringBuffer`.
9. `stringBuffer.deleteCharAt(10);`: This line deletes the character at position 10 in the `stringBuffer`.
10. `stringBuffer.reverse();`: This line reverses the content of the `stringBuffer`.
11. `String result = stringBuffer.toString();`: This line converts the `stringBuffer` to a `String` and assigns it to the variable `result`.
12. `int length = stringBuffer.length();`: This line gets the length (number of characters) of the `stringBuffer`.
13. `int capacity = stringBuffer.capacity();`: This line gets the capacity of the `stringBuffer`.
14. The following lines print the results to the console:

- `"Modified String: " + result`: Prints the modified string.
- `"Length of StringBuffer: " + length`: Prints the length of the `stringBuffer`.
- `"Capacity of StringBuffer: " + capacity`: Prints the capacity of the `stringBuffer`.

The program demonstrates various operations that can be performed on a `StringBuffer` object, including appending, inserting, replacing, setting, deleting, and reversing characters, as well as converting it to a `String` and retrieving its length and capacity.

Regenerate