Object Oriented Programming Sample Paper

Problem 1: Smart Home Device Management Using Virtual Functions

You are developing a **Smart Home Device Management System** where multiple types of smart devices (e.g., Smart Lights, Smart Locks, and Smart Thermostats) are controlled. The system should use **virtual functions** to ensure each device type implements its own method of displaying information dynamically.

Create the following **class hierarchy**:

- **SmartDevice (Base Class - Abstract Class)**

  Stores `device ID` and provides a **pure virtual function** `displayInfo()`.

- **SmartLight (Derived from SmartDevice)**

  Stores **brightness level** (0–100%) and overrides `displayInfo()`.

- **SmartLock (Derived from SmartDevice)**

  Stores **lock status** (Locked/Unlocked) and overrides `displayInfo()`.

- **SmartThermostat (Derived from SmartDevice)**

Stores **current temperature** (in °C) and overrides `displayInfo()`.

Your program should allow **inputting multiple Smart Devices** and display them **dynamically using virtual functions**.

Input Format:

- The first line contains an integer `N` (number of smart devices).
- The next `N` lines contain details in the format:

```
device_type device_ID additional_info
```

Where:

- `device_type`: `"Light"`, `"Lock"`, or `"Thermostat"`
- `device_ID`: Alphanumeric string (max length 10)
- `additional_info`:
  - If `Light`, enter **brightness level** (0–100%)
  - If `Lock`, enter `"Locked"` or `"Unlocked"`
  - If `Thermostat`, enter **temperature** in Celsius

Output Format:

For each Smart Device, print:

```
Smart Device Details:
Device ID: <device_ID>
<Specific Details>
```

Each device's details should be printed separately.

Constraints:

- `1 ≤ N ≤ 100`
- `device_ID` is an alphanumeric string (max length 10).
- **brightness level**: `0 ≤ level ≤ 100`
- **lock status**: `"Locked"` or `"Unlocked"`

- **temperature:** `50 ≤ temperature ≤ 50 (in °C)`

Sample Test Cases:

Test Case 1: Basic Test Case

Input:

```
1
Light D123 75
```

Output:

```
Smart Device Details:
Device ID: D123
Brightness Level: 75%
```

Explanation:

- The smart light `D123` has a brightness level of 75%.

Test Case 2: Multiple Devices

Input:

```
3
Lock S567 Locked
Thermostat T890 22
Light L345 40
```

Output:

```
Smart Device Details:
Device ID: S567
Lock Status: Locked
Smart Device Details:
Device ID: T890
Current Temperature: 22°C
Smart Device Details:
```

```
Device ID: L345
Brightness Level: 40%
```

Explanation:

- The `S567` smart lock is Locked.
- The `T890` smart thermostat is set at 22°C.
- The `L345` smart light is at 40% brightness.

🎉 Solution:

Problem 2: Robotics Control System using Virtual Base Class

You are designing a **Robotics Control System** that categorizes robots based on their **functionality** and **energy source**. The system should use **virtual base classes** to avoid redundant data.

Class Hierarchy to Create:

- **Robot** (Base Class - *Virtual*):

  Stores:

  - `model_name`
  - `weight`

- **Functionality** (Derived from Robot):

  Stores:

  - `task_type` (e.g., Industrial, Medical, Military)

- **EnergySource** (Derived from Robot):

  Stores:

  - `energy_type` (e.g., Electric, Solar, Nuclear)

- **SmartRobot** (Derived from **Functionality** and **EnergySource**):

  Stores:

  - `AI_enabled` flag (Yes or No)

    Displays:

  - Complete robot details

Note: Your program should allow inputting multiple **SmartRobot** details and display them.

Input Format:

1. The first line contains an integer N – the number of robots.
2. The next **N** lines contain robot details in the format:

```
model_name weight task_type energy_type AI_enabled
```

Where:

- `model_name`: Alphanumeric (max length 20)
- `weight`: Integer in kg
- `task_type`: One of `Industrial`, `Medical`, `Military`, `Exploration`
- `energy_type`: One of `Electric`, `Solar`, `Nuclear`
- `AI_enabled`: Either `Yes` or `No`

Output Format:

For each **Smart Robot**, print the following:

```
Smart Robot Details:
Model Name: <model_name>
Weight: <weight> kg
Task Type: <task_type>
Energy Source: <energy_type>
AI Enabled: <Yes/No>
```

Each robot's details should be printed **separately**.

Constraints:

- $1 \leq N \leq 100$
- $1 \leq weight \leq 1000$ Kg
- `model_name`: Alphanumeric string (max length 20)
- `task_type`: Must be one of the predefined options

- `energy_type`: Must be one of the predefined options
- `AI_enabled`: Yes / No

Sample Test Cases:

Test Case 1 (Basic Test Case)

Input:

```
1
XJ-900 250 Industrial Electric Yes
```

Output:

```
Smart Robot Details:
Model Name: XJ-900
Weight: 250 kg
Task Type: Industrial
Energy Source: Electric
AI Enabled: Yes
```

Explanation:

- The robot XJ-900 weighs 250 kg, is used in Industrial tasks, runs on Electric power, and has AI capability.

Test Case 2 (Multiple Robots)

Input:

```
2
RX-500 150 Medical Solar No
WQ-2000 500 Military Nuclear Yes
```

Output:

```
Smart Robot Details:
Model Name: RX-500
Weight: 150 kg
Task Type: Medical
```

```
Energy Source: Solar
AI Enabled: No
Smart Robot Details:
Model Name: WQ-2000
Weight: 500 kg
Task Type: Military
Energy Source: Nuclear
AI Enabled: Yes
```

Solution

Problem 3: Smart Home Appliance Control using Virtual Base Class

You need to implement a **Smart Home Appliance Control System** using **Virtual Base Class** in C++.

Create a base class `Appliance` that stores common attributes like:

- `brand`
- `power` consumption (in watts)

Then, create two intermediate derived classes:

- `WiredDevice`: Stores `voltage` rating (in volts)
- `WirelessDevice`: Stores `network_type` (WiFi/Bluetooth)

Finally, create a derived class `SmartAppliance` that inherits both `WiredDevice` and `WirelessDevice`. This class should avoid multiple copies of `Appliance` attributes using **virtual inheritance**.

Your program should allow users to **enter smart appliance details** and **display them**.

Input Format

- The first line contains an integer `N` (number of smart appliances).
- Each of the next `N` lines contains details in the format:

  ```
  brand power voltage network_type
  ```

Output Format

For each Smart Appliance, print:

```
Smart Appliance Details:
Brand: <brand>
Power: <power> W
Voltage: <voltage> V
Network Type: <network_type>
```

Each appliance's details should be printed **separately**.

📌 Constraints

- 1≤N≤100 (Number of appliances)
- 10≤power≤5000 (in watts)
- 110≤voltage≤240 (voltage rating)
- `network_type` can be **WiFi** or **Bluetooth** only.
- `brand` contains only **lowercase English letters** (max length 20).

✅ Sample Test Cases

Test Case 1 (Basic Test Case)

Input:

```
1
samsung 1000 220 WiFi
```

Output:

```
Smart Appliance Details:
Brand: samsung
Power: 1000 W
Voltage: 220 V
Network Type: WiFi
```

Explanation:

- The given appliance is a **Samsung** device that consumes **1000 watts** of power.
- It operates on **220V** and connects via **WiFi**.

Test Case 2 (Multiple Appliances)

Input:

```
2
lg 1500 110 Bluetooth
sony 2000 240 WiFi
```

Output:

```
Smart Appliance Details:
Brand: lg
Power: 1500 W
Voltage: 110 V
Network Type: Bluetooth

Smart Appliance Details:
Brand: sony
Power: 2000 W
Voltage: 240 V
Network Type: WiFi
```

🕯️ Solution

Problem 4: Smart Payment System Using Polymorphism

You are developing a Smart Payment System that supports multiple payment methods (Credit Card, Digital Wallet, and UPI). The system should use polymorphism to handle different payment methods dynamically.

Create the following class hierarchy:

1. **Payment** (Base Class - Abstract Class): Stores transaction ID and provides a pure virtual function `processPayment()`.

2. **CreditCard** (Derived from Payment): Stores card number (last 4 digits) and overrides `processPayment()`.
3. **DigitalWallet** (Derived from Payment): Stores wallet provider (PayPal, Paytm, Google Pay, etc.) and overrides `processPayment()`.
4. **UPI** (Derived from Payment): Stores UPI ID and overrides `processPayment()`.

Your program should allow inputting multiple payment transactions and display their details dynamically using polymorphism.

Input Format:

- The first line contains an integer `N` (number of payment transactions).
- The next `N` lines contain details in the format:

```
payment_type transaction_ID additional_info
```

- `payment_type`: "CreditCard", "DigitalWallet", or "UPI"
- `transaction_ID`: Alphanumeric string (max length 12)
- `additional_info`:
  - If `CreditCard`, enter last 4 digits of the card.
  - If `DigitalWallet`, enter wallet provider (PayPal, Paytm, Google Pay, etc.).
  - If `UPI`, enter UPI ID.

Output Format:

For each Payment Transaction, print:

```
Payment Details:
Transaction ID: <transaction_ID>
<Specific Details>
```

Each transaction's details should be printed separately.

Constraints:

- `1 ≤ N ≤ 100`

- `transaction_ID` is an alphanumeric string (max length 12).
- `card number`: 4-digit number (`1000 ≤ number ≤ 9999`).
- `wallet provider`: Alphanumeric string (max length 15).
- `UPI ID`: Alphanumeric string (max length 20).

Sample Test Cases

Test Case 1 (Basic Test Case)

Input:

```
1
CreditCard TXN123 5678
```

Output:

```
Payment Details:
Transaction ID: TXN123
Credit Card Last 4 Digits: 5678
```

Explanation:

- The payment `TXN123` was made using a Credit Card with last 4 digits `5678`.

Test Case 2 (Multiple Transactions)

Input:

```
3
UPI TXN789 abc@upi
DigitalWallet TXN456 Paytm
CreditCard TXN999 4321
```

Output:

```
Payment Details:
Transaction ID: TXN789
UPI ID: abc@upi
```

```
Payment Details:
Transaction ID: TXN456
Digital Wallet Provider: Paytm

Payment Details:
Transaction ID: TXN999
Credit Card Last 4 Digits: 4321
```

 Solution

---

Problem 5: Vehicle Management System Using Inheritance

You need to design a Vehicle Management System using hierarchical inheritance. Implement a base class `Vehicle` that defines common vehicle properties. Then, derive two subclasses:

- **Car**: Represents a car with attributes: model name, fuel capacity (in liters), mileage (km per liter), and seating capacity.
- **Bike**: Represents a bike with attributes: model name, engine capacity (cc), mileage (km per liter), and type (sports/cruiser).

Your program should allow users to store vehicle details and display them.

Input Format:

- The first line contains an integer `N` (number of vehicles).
- Each of the next `N` lines contains:
  - `"car model fuel_capacity mileage seating_capacity"` (for cars)
  - `"bike model engine_capacity mileage type"` (for bikes)

Output Format:

1. For each **Car**, output:

```
Vehicle Type: Car
Model: <model>
Fuel Capacity: <fuel_capacity> L
Mileage: <mileage> km/l
Seating Capacity: <seating_capacity>
```

2. For each **Bike**, output:

```
Vehicle Type: Bike
Model: <model>
Engine Capacity: <engine_capacity> cc
Mileage: <mileage> km/l
Type: <type>
```

Each vehicle's details should be printed separately.

Constraints:

- `1 ≤ N ≤ 100` (Number of vehicles)
- `5 ≤ fuel_capacity ≤ 100` (for cars)
- `50 ≤ engine_capacity ≤ 2000` (for bikes)
- `5 ≤ mileage ≤ 50` (for both cars and bikes)
- `2 ≤ seating_capacity ≤ 7` (for cars)
- `type` can be either "sports" or "cruiser" (for bikes).
- Model names contain only lowercase English letters and numbers.

Sample Test Cases

Test Case 1 (Basic Test with One Car and One Bike)

Input:

```
2
car swift 45 20 5
bike ninja 650 25 sports
```

Output:

```
Vehicle Type: Car
Model: swift
Fuel Capacity: 45 L
Mileage: 20 km/l
Seating Capacity: 5

Vehicle Type: Bike
```

```
Model: ninja
Engine Capacity: 650 cc
Mileage: 25 km/l
Type: sports
```

Explanation:

- Car `swift` has fuel capacity 45L, mileage 20 km/l, seating for 5.
- Bike `ninja` has engine capacity 650cc, mileage 25 km/l, type sports.

Test Case 2 (Multiple Vehicles)

Input:

```
3
car tesla 80 30 5
bike ducati 1200 18 sports
car bmw 60 22 4
```

Output:

```
Vehicle Type: Car
Model: tesla
Fuel Capacity: 80 L
Mileage: 30 km/l
Seating Capacity: 5

Vehicle Type: Bike
Model: ducati
Engine Capacity: 1200 cc
Mileage: 18 km/l
Type: sports

Vehicle Type: Car
Model: bmw
Fuel Capacity: 60 L
Mileage: 22 km/l
Seating Capacity: 4
```

📣 Solution

Problem 6: Advanced Shape Hierarchy with Multiple Inheritance

Design a class hierarchy to represent different types of shapes using multiple inheritance. Implement a base class `Shape` with a pure virtual function `area()`. Derive two classes, `Rectangle` and `Circle`, from `Shape`. Additionally, create a class `Colored` that adds a color attribute. Finally, derive classes `ColoredRectangle` and `ColoredCircle` that inherit from both `Rectangle` and `Colored`, and `Circle` and `Colored`, respectively.

The program should read the details of multiple shapes from input, calculate their area, and display their properties, including color, dimensions, and computed area.

Input Format:

- The first line contains an integer **N** (number of shapes).
- The next **N** lines contain details of each shape in the following format:
  - `"rectangle width height color"` (for rectangles)
  - `"circle radius color"` (for circles)

Output Format:

For each shape, output:

```
Shape: <Type>, Color: <Color>
Dimensions of the shape:
For Rectangles: Width: <width>, Height: <height>
For Circles: Radius: <radius>
Area: <computed area> (formatted to two decimal places)
```

Each shape's details should be printed separately.

Constraints:

- 1≤N≤1001 \leq N \leq 100 (Number of shapes)

- 1.0≤width,height,radius≤1000.01.0 \leq \text{width}, \text{height}, \text{radius} \leq 1000.0
- Color is a non-empty string containing only lowercase English letters.

Sample Test Cases:

Test Case 1 (Basic Test with One Rectangle and One Circle):

Input:

```
2
rectangle 5.0 10.0 green
circle 4.0 yellow
```

Output:

```
Shape: Rectangle, Color: green
Width: 5.00, Height: 10.00
Area: 50.00
Shape: Circle, Color: yellow
Radius: 4.00
Area: 50.27
```

Explanation:

- A rectangle with width = 5.0 and height = 10.0 has an area of 50.00 (5 × 10 = 50.00).
- A circle with radius = 4.0 has an area of approximately 50.27 (π×42≈50.27\pi \times 4^2 \approx 50.27).

Test Case 2 (Multiple Rectangles with Different Colors):

Input:

```
3
rectangle 3.5 2.0 red
rectangle 6.0 7.5 blue
circle 2.5 orange
```

Output:

```
Shape: Rectangle, Color: red
Width: 3.50, Height: 2.00
Area: 7.00
Shape: Rectangle, Color: blue
Width: 6.00, Height: 7.50
Area: 45.00
Shape: Circle, Color: orange
Radius: 2.50
Area: 19.63
```

Explanation:

- The first rectangle has width = 3.5 and height = 2.0, so its area is 7.00 (3.5 × 2.0).
- The second rectangle has width = 6.0 and height = 7.5, so its area is 45.00 (6.0 × 7.5).
- The circle has radius = 2.5, so its area is approximately 19.63 (π×2.52≈19.63\pi \times 2.5^2 \approx 19.63).

Test Case 3 (Edge Case with Large Values):

Input:

```
2
rectangle 1000.0 500.0 black
circle 100.0 white
```

Output:

```
Shape: Rectangle, Color: black
Width: 1000.00, Height: 500.00
Area: 500000.00
Shape: Circle, Color: white
Radius: 100.00
Area: 31415.93
```

🥁 Solution

Problem 7: Vehicle Management System Using Inheritance

You need to design a Vehicle Management System using hierarchical inheritance. Implement a base class `Vehicle` that defines common vehicle properties. Then, derive two subclasses:

- **Car**: Represents a car with attributes:

  `model name`, `fuel capacity` (in liters), `mileage` (km per liter), and `seating capacity`.

- **Bike**: Represents a bike with attributes:

  `model name`, `engine capacity` (cc), `mileage` (km per liter), and `type` (`sports/cruiser`).

Your program should allow users to store vehicle details and display them.

Input Format:

- The first line contains an integer `N` (number of vehicles).
- Each of the next `V` lines contains:

  `"car model fuel_capacity mileage seating_capacity"` (for cars)

  `"bike model engine_capacity mileage type"` (for bikes)

Output Format:

1. For each Car, output:

```
Vehicle Type: Car
Model: <model>
Fuel Capacity: <fuel_capacity> L
Mileage: <mileage> km/l
Seating Capacity: <seating_capacity>
```

2. For each Bike, output:

```
Vehicle Type: Bike
Model: <model>
Engine Capacity: <engine_capacity> cc
```

```
Mileage: <mileage> km/l
Type: <type>
```

Each vehicle's details should be printed separately.

Constraints:

- 1≤N≤100 *(Number of vehicles)*
- 5≤fuel_capacity≤100 (*for cars)*
- 50≤engine_capacity≤2000 *(for bikes)*
- 5≤mileage≤50 *(for both cars and bikes)*
- 2≤seating_capacity≤7 *(for cars)*
- `type` can be either `"sports"` or `"cruiser"` *(for bikes)*
- Model names contain only lowercase English letters and numbers.

Sample Test Cases:

Test Case 1 (Basic Test with One Car and One Bike)

Input:

```
2
car swift 45 20 5
bike ninja 650 25 sports
```

Output:

```
Vehicle Type: Car
Model: swift
Fuel Capacity: 45 L
Mileage: 20 km/l
Seating Capacity: 5

Vehicle Type: Bike
Model: ninja
Engine Capacity: 650 cc
```

```
Mileage: 25 km/l
Type: sports
```

Explanation:

Car `swift` has fuel capacity `45L`, mileage `20 km/l`, seating for `5`.

Bike `ninja` has engine capacity `650cc`, mileage `25 km/l`, type `sports`.

Test Case 2 (Multiple Vehicles)

Input:

```
3
car tesla 80 30 5
bike ducati 1200 18 sports
car bmw 60 22 4
```

Output:

```
Vehicle Type: Car
Model: tesla
Fuel Capacity: 80 L
Mileage: 30 km/l
Seating Capacity: 5

Vehicle Type: Bike
Model: ducati
Engine Capacity: 1200 cc
Mileage: 18 km/l
Type: sports

Vehicle Type: Car
Model: bmw
Fuel Capacity: 60 L
Mileage: 22 km/l
Seating Capacity: 4
```

Explanation:

Car `tesla` has `80L` fuel capacity, `30 km/l` mileage, `5 seats`.

Bike `ducati` has `1200cc` engine, `18 km/l` mileage, type `sports`.

Car `bmw` has `60L` fuel capacity, `22 km/l` mileage, `4 seats`.

Test Case 3 (Edge Case with Maximum Constraints)

Input:

```
2
bike pulsar 2000 50 cruiser
car ford 100 10 7
```

Output:

```
Vehicle Type: Bike
Model: pulsar
Engine Capacity: 2000 cc
Mileage: 50 km/l
Type: cruiser

Vehicle Type: Car
Model: ford
Fuel Capacity: 100 L
Mileage: 10 km/l
Seating Capacity: 7
```

Explanation:

Bike `pulsar` has maximum engine capacity (`2000cc`) and highest mileage (`50 km/l`).

Car `ford` has maximum fuel capacity (`100L`) and seating capacity (`7 seats`).

🧯 Solution

Problem 8: Appliance Inventory System

You're building an inventory system for a home appliance store. The store has two types of appliances: **Washing Machines** and **Refrigerators**.

Each appliance has a **model name**. You need to store their respective details and then display the information for all appliances entered.

Input Format

- First line: An integer `N` (number of appliances)
- Next `N` lines: For each appliance:
  - First value: Appliance type (`washing_machine` or `refrigerator`)
  - Next values:
    - For `washing_machine`: `model_name drum_size wash_modes spin_speed`
    - For `refrigerator`: `model_name capacity door_type energy_rating`

Constraints

```
1 ≤ N ≤ 100 (Number of appliances)

5 ≤ drum_size ≤ 20 (in kg, for washing machines)
2 ≤ wash_modes ≤ 15 (for washing machines)
500 ≤ spin_speed ≤ 2000 (in RPM, for washing machines)

100 ≤ capacity ≤ 600 (in liters, for refrigerators)
door_type ∈ { "single", "double" } (for refrigerators)
1 ≤ energy_rating ≤ 5 (stars)
```

Model names contain only lowercase English letters and numbers.

Output Format

For each appliance, print the following info:

- Appliance Type
- Model Name
- (Washing Machine): Drum Size, Wash Modes, Spin Speed
- (Refrigerator): Capacity, Door Type, Energy Rating

Sample Input

```
3
washing_machine whirlpoolx 7 10 1200
```

```
refrigerator samsungcool 350 double 4
washing_machine lgsmartwash 8 12 1400
```

Sample Output

```
Appliance Type: Washing Machine
Model: whirlpoolx
Drum Size: 7 kg
Wash Modes: 10
Spin Speed: 1200 RPM

Appliance Type: Refrigerator
Model: samsungcool
Capacity: 350 L
Door Type: double
Energy Rating: 4 stars

Appliance Type: Washing Machine
Model: lgsmartwash
Drum Size: 8 kg
Wash Modes: 12
Spin Speed: 1400 RPM
```

 Solution

Problem 9: Electronic Gadget Inventory System

You're managing inventory for a store that sells two types of gadgets: **Laptops** and **Smartphones**.

Each gadget has a model name and other respective attributes. Your task is to store the details and display them.

Input Format

- First line: An integer `N` (number of gadgets)
- Next `N` lines: For each gadget:

- First value: Gadget type (`laptop` or `smartphone`)
- Next values:
  - For `laptop`: `model_name ram_size storage_size battery_backup`
  - For `smartphone`: `model_name camera_megapixel screen_size battery_capacity`

Constraints

```
1 ≤ N ≤ 100

4 ≤ ram_size ≤ 64 (in GB)
128 ≤ storage_size ≤ 2048 (in GB)
2 ≤ battery_backup ≤ 12 (in hours)

8 ≤ camera_megapixel ≤ 108
4 ≤ screen_size ≤ 7 (in inches)
2000 ≤ battery_capacity ≤ 6000 (in mAh)
```

Model names contain only lowercase English letters and numbers.

Sample Input

```
2
laptop dellxps 16 512 10
smartphone redmi12 50 6 5000
```

Sample Output

```
Gadget Type: Laptop
Model: dellxps
RAM Size: 16 GB
Storage: 512 GB
Battery Backup: 10 hours

Gadget Type: Smartphone
Model: redmi12
Camera: 50 MP
```

```
Screen Size: 6 inches
Battery Capacity: 5000 mAh
```

🎄 Solution

Problem 10: Course Management System

Create a system to manage two types of courses: **Online** and **Offline**.

Input Format

- First line: An integer `N` (number of courses)
- Next `N` lines: For each course:
  - First value: Course type (`online` or `offline`)
  - Next values:
    - For `online`: `title duration platform`
    - For `offline`: `title duration location room_number`

Constraints

```
1 ≤ N ≤ 100

1 ≤ duration ≤ 52 (in weeks)

platform ∈ { "coursera", "edx", "udemy", "udacity" }
location: only lowercase English characters
room_number: string of digits and letters (e.g., A101, B5, etc.)
```

Course title contains only lowercase letters and underscores.

Sample Input

```
2
online python_basics 6 coursera
offline cpp_advance 10 buildinga A101
```

Sample Output

```
Course Type: Online
Title: python_basics
Duration: 6 weeks
Platform: coursera

Course Type: Offline
Title: cpp_advance
Duration: 10 weeks
Location: buildinga
Room: A101
```

 Solution