Day 6 - OOPs, Access Modifiers & Encapsulation

Easy & Simple Notes for Beginners

🎯 What We'll Learn Today

- Object-Oriented Programming (OOP) basics
- Classes and Objects
- Constructor and Destructor
- Access Modifiers (Public, Protected, Private)
- Encapsulation with real-life examples

🔵 1. What is OOP?

Definition: OOP (Object-Oriented Programming) is a way to design code using real-life objects and classes.

Logic: Instead of writing separate functions, we group related data and functions together in a "class".

Real-life Example:

- Think of a **Car** - it has properties (color, speed) and actions (start, stop)
- Think of a **Remote** - it has buttons (volume, power) and functions (increase volume, turn on TV)

Benefits:

- Code is more organized
- Easy to understand and maintain
- Reusable code

📚 2. Basic OOP Concepts

Term	Meaning	Example
Class	Blueprint for creating objects	Car blueprint
Object	Instance of a class	Your actual car
Constructor (`__init__`)	Special method that runs when object is created	Setting up the car
Destructor (`__del__`)	Special method that runs when object is destroyed	Scrapping the car
Self	Refers to the object itself	"This car"

🧬 3. Creating Your First Class

Definition: A class is like a template or blueprint.

Example:

python

```python
class IITM:
    def intro(self):
        print('Hello Buddy!!, how are you??')

    def say_hello(self):
        print('Hello!!')

    def bye(self):
        print('Bye Bye!!')

# Create object
ayan = IITM()
print(type(ayan))  # Shows: <class '__main__.IITM'>
```

Logic:

- `class IITM:` creates a class template
- `ayan = IITM()` creates an object from the class
- `self` refers to the specific object

🎮 4. Real-Life Example - TV Remote

Definition: A remote control has buttons (data) and functions (methods).

Example:

python

```
class Remote:
    def __init__(self):
        print('Remote Created ✅')

    def volume_up(self):
        print('Volume up by 1 unit')

    def volume_down(self):
        print('Volume down by 1 unit')

    def power_on(self):
        print('TV is ON')

    def power_off(self):
        print('TV is OFF')

    def __del__(self):
        print('Remote Destroyed ❌')

# Using the remote
mi = Remote()
mi.power_on()
mi.volume_up()
del mi
```

Logic:

- `__init__()` runs when remote is created
- `__del__()` runs when remote is destroyed
- Each method does a specific remote function

🔐 5. Access Modifiers

Definition: Access modifiers control who can access variables and methods.

Types:

Type	Syntax	Who can access?	Example
Public	`variable`	Everyone	`self.pin = 1234`
Protected	`_variable`	Within class & subclasses	`self._balance = 5000`
Private	`__variable`	Only inside class	`self.__secret = 9999`

Example:

python

```python
class ATM:
    def __init__(self):
        self.pin = 1234        # Public - anyone can access
        self._balance = 5000   # Protected - use carefully
        self.__bank_name = "SBI"  # Private - hidden from outside

    def show(self):
        print(f"PIN: {self.pin}")
        print(f"Balance: {self._balance}")
        print(f"Bank: {self.__bank_name}")

atm = ATM()
print(atm.pin)              # ✅ Works - Public
print(atm._balance)         # ⚠️ Works but not recommended
print(atm._ATM__bank_name)  # ✅ Access private using name mangling
```

Logic:

- Public: Like your name - everyone can know
- Protected: Like your phone number - share carefully
- Private: Like your PIN - keep secret

🔒 6. Encapsulation

Definition: Bundling data and methods in one unit (class) and hiding internal details.

Real-life Example: When you use an ATM:

- **Data:** Your PIN, Balance, Account Number
- **Methods:** Withdraw, Check Balance, Change PIN
- You don't see the internal bank code - that's encapsulation!

Example:

python

```python
class BankAccount:
    def __init__(self, name, balance):
        self.name = name            # Public
        self._account_no = 12345    # Protected
        self.__balance = balance    # Private

    def deposit(self, amount):
        self.__balance += amount
        print(f"Deposited ₹{amount}. New balance: ₹{self.__balance}")

    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew ₹{amount}. Remaining: ₹{self.__balance}")
        else:
            print("Insufficient balance!")

    def check_balance(self):
        return self.__balance

# Usage
account = BankAccount("Rahul", 10000)
account.deposit(2000)
account.withdraw(500)
print(f"Current balance: ₹{account.check_balance()}")
```

Logic:

- Balance is private - can't be directly changed
- Only through deposit/withdraw methods
- This prevents accidental changes to balance

🚙 7. Practical Example - Car Class

Definition: A car has properties (color, wheels) and actions (start, speed up).

Example:

python

```python
class Car:
    def __init__(self, brand, color):
        self.brand = brand      # Public
        self.color = color      # Public
        self.wheels = 4         # Public
        self.__engine_on = False  # Private
        print(f'Your {color} {brand} Car is Ready!')

    def start_engine(self):
        self.__engine_on = True
        print('Engine Started 🚗')

    def stop_engine(self):
        self.__engine_on = False
        print('Engine Stopped')

    def speed_up(self):
        if self.__engine_on:
            speed = 0
            while speed <= 100:
                speed += 20
                print(f'Speed: {speed} km/hr')
                if speed >= 80:
                    print('Drive Safe! Speed Limit ahead 🚦')
                    break
        else:
            print('Start the engine first!')

# Usage
bmw = Car("BMW", "Black")
bmw.start_engine()
bmw.speed_up()
```

Logic:

- Constructor sets up the car
- Engine state is private (can't directly change)
- Methods control car behavior safely

🎯 Key Takeaways

✅ **OOP** makes code more organized and reusable

✅ **Classes** are blueprints, **Objects** are real instances

✅ **Constructor** (`__init__`) sets up objects when created

✅ **Destructor** (`__del__`) cleans up when objects are destroyed

✅ **Access Modifiers** control who can access data:

- Public: Everyone can access
- Protected: Use carefully
- Private: Hidden from outside

✅ **Encapsulation** bundles data and methods together safely

✅ **Real-life modeling** makes programming easier to understand

---

📝 Practice Questions

Question 1: Basic Class

Create a `Student` class with:

- Name and age as public variables
- A method `introduce()` that prints student details

Answer:

```python
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        print(f"Hi, I'm {self.name} and I'm {self.age} years old")

# Usage
student1 = Student("Amit", 20)
student1.introduce()
```

Question 2: Access Modifiers

Create a `Phone` class with:

- Public: brand, model
- Protected: price
- Private: imei_number

Answer:

```python
class Phone:
    def __init__(self, brand, model, price, imei):
        self.brand = brand          # Public
        self.model = model          # Public
        self._price = price         # Protected
        self.__imei_number = imei   # Private

    def show_details(self):
        print(f"Brand: {self.brand}")
        print(f"Model: {self.model}")
        print(f"Price: ₹{self._price}")
        print(f"IMEI: {self.__imei_number}")

# Usage
phone = Phone("Samsung", "Galaxy S21", 50000, "123456789012345")
phone.show_details()
```

Question 3: Encapsulation

Create a `Calculator` class with private result variable and public methods for operations.

Answer:

```python
```

```python
class Calculator:
    def __init__(self):
        self.__result = 0  # Private

    def add(self, num):
        self.__result += num
        return self.__result

    def subtract(self, num):
        self.__result -= num
        return self.__result

    def multiply(self, num):
        self.__result *= num
        return self.__result

    def get_result(self):
        return self.__result

    def reset(self):
        self.__result = 0

# Usage
calc = Calculator()
calc.add(10)
calc.multiply(5)
print(f"Result: {calc.get_result()}")  # Output: 50
```

🏆 Assignment Questions

Assignment 1: Fan Class

Create a `Fan` class with:

- Speed levels (0-5)
- Methods: turn_on(), turn_off(), increase_speed(), decrease_speed()
- Use encapsulation to protect speed variable

Assignment 2: Employee Class

Create an `Employee` class with:

- Name, salary, and department
- Methods: get_details(), increase_salary(percentage)
- Use appropriate access modifiers

Assignment 3: Bank Account

Create a `BankAccount` class with:

- Account holder name, account number, balance
- Methods: deposit(), withdraw(), check_balance()
- Use encapsulation to protect balance
- Add validation for withdrawal (sufficient balance check)

Assignment 4: Library Book

Create a `Book` class with:

- Title, author, price, and availability status
- Methods: borrow_book(), return_book(), get_book_info()
- Use private variables for availability status

💡 Remember

- **Start simple** - begin with basic classes
- **Use real-life examples** - car, phone, ATM
- **Practice access modifiers** - public, protected, private
- **Encapsulation is important** - hide internal details
- **Constructor sets up objects** - use `__init__()`
- **Think in terms of objects** - not just functions

Happy Learning! 🚀