🐍 Session 17-18: OOP Advanced Concepts

🎭 Topics: Inheritance, Polymorphism, Encapsulation

🎯 Learning Goals

By the end of this session, you will be able to:

- ✔ Use **Inheritance** to reuse and extend code.
- ✔ Apply **Polymorphism** to write flexible code.
- ✔ Understand **Encapsulation** and use access modifiers.

🧬 1. Inheritance

Definition:
Inheritance allows one class (**child**) to use properties and methods of another class (**parent**).
This helps in **reusability** and makes code shorter & cleaner.

💡 Analogy:

- A **child** inherits traits (like eyes 👀, hair 💇‍♂️) from their **parents**.
- But they can also have **their own unique features**.

Example: Single Inheritance

```
# Parent class
class Animal:
    def speak(self):
        print("This animal makes a sound 🐾")

# Child class
class Dog(Animal):
    def speak(self):
        print("Woof! 🐶")

d = Dog()
d.speak()
```

🔎 Explanation:

Dog automatically gets everything from Animal.

But Dog can also override the method with its own behavior.

Types of Inheritance (just concepts here)

- Single → One parent, one child
- Multiple → Child inherits from more than one parent
- Multilevel → Child → Parent → Grandparent chain
- Hierarchical → Many children inherit from same parent
- Hybrid → Mix of the above

Example: Method Overriding (Run-time Polymorphism)

```
class Bird:
    def fly(self):
        print("Some birds can fly 🐦")

class Penguin(Bird):
    def fly(self):
        print("Penguins cannot fly 🐧")

b = Bird()
p = Penguin()

b.fly()
p.fly()
```

🎭 2. Polymorphism

Definition: Polymorphism = many forms. The same function name can work in different ways depending on context.

💡 Analogy: The word "run" means different things:

- You can run 🏃 in the park.
- A program can run 💻 on your computer.
- A car can run 🚗 on petrol.

Example: Duck Typing (Python Special Polymorphism 🦆)

```
class Duck:
    def sound(self):
        print("Quack Quack 🦆")

class Dog:
    def sound(self):
```

```
        print("Woof Woof 🐶")

def make_sound(animal):
    animal.sound()

make_sound(Duck())
make_sound(Dog())
```

In Python, if it walks like a duck and quacks like a duck → we don't care about type, we just care if it has .sound().

🔐 3. Encapsulation

Definition: Encapsulation = wrapping data and methods into a single unit (class). We also use access modifiers to control access to data.

💡 Analogy:

Think of a capsule pill 💊 → it hides the medicine inside.

A bank account hides your balance but allows controlled actions like deposit() and withdraw().

Access Modifiers in Python

- Public → Accessible everywhere (self.name)
- Protected → Meant for internal use (self._name)
- Private → Cannot be accessed directly (self.__balance)

```
class BankAccount:
    def __init__(self, name, balance):
        self.name = name        # public
        self._account_no = 1234 # protected
        self.__balance = balance # private

    def deposit(self, amount):
        self.__balance += amount
        print("Deposited:", amount)

    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
            print("Withdrew:", amount)
        else:
            print("Insufficient funds ❌")

    def display_balance(self):
        print("Balance:", self.__balance)
```

```
acc = BankAccount("Rahul", 5000)
acc.deposit(2000)
acc.withdraw(1000)
acc.display_balance()

# Try accessing directly
print(acc.name)         # ✅ works
print(acc._account_no)  # ⚠️ works but not recommended
# print(acc.__balance)  # ❌ error: private
```

📝 Practice Problems

👉 Q1. Create a class Shape with method area().

Create subclasses Rectangle and Circle and override area() method.

👉 Q2. Create a base class Employee.

Inherit Manager and Developer from it.

Override a method show_role().

👉 Q3. Create a class SecureData with private variable.

Provide getter and setter methods to access/update the private variable safely.