Day 7 - OOP: Inheritance, Polymorphism, Abstraction

Easy & Simple Notes for Beginners

🎯 What We'll Learn Today

- Inheritance (Single, Multiple, Multilevel, Hybrid)
- Polymorphism (Method Overriding & Overloading)
- Abstraction (Abstract Base Classes)
- Real-world examples with practical code
- Complete OOP concepts recap

🔗 1. Inheritance

Definition: Inheritance allows a new class (child) to acquire properties and methods from an existing class (parent).

Real-life Example: A child inherits traits from parents - eye color, height, etc.

Benefits:

- **Code Reuse:** Don't write same code again
- **Maintainability:** Easy to update and fix
- **Scalability:** Easy to add new features

Basic Syntax:

```
python
```

```
class Parent:
    def parent_method(self):
        print("I'm from parent")

class Child(Parent):
    def child_method(self):
        print("I'm from child")

# Child can use both methods
obj = Child()
obj.parent_method()  # Inherited from parent
obj.child_method()   # Own method
```

🏗 2. Types of Inheritance

2.1 Single Inheritance

Definition: One parent, one child.

Example:

```
python
```

```
class BMW:  # Parent class
    def super_turbo_engine(self):
        print('Super Turbo Engine!')

class Audi(BMW):  # Child class inherits from BMW
    def auto_pilot(self):
        print('Auto Pilot Mode!')

# Usage
audi_r8 = Audi()
audi_r8.auto_pilot()          # Own method
audi_r8.super_turbo_engine()  # Inherited method
```

Logic: Audi gets BMW's engine technology + adds its own features.

2.2 Multiple Inheritance

Definition: Child class inherits from multiple parents.

Example:

```
python
```

```
class JBL:
    def speaker(self):
        print('High Quality Sound')

class Sony:
    def camera(self):
        print('4K Camera')

class Mobile(JBL, Sony):  # Inherits from both
    def apps(self):
        print('Mobile Apps')

# Usage
m = Mobile()
m.speaker()  # From JBL
m.camera()   # From Sony
m.apps()     # Own method
```

Logic: Mobile gets sound from JBL + camera from Sony + adds its own apps.

2.3 Multilevel Inheritance

Definition: Chain of inheritance - grandparent → parent → child.

Example:

```
python
```

```
class GrandFather:
    def land(self):
        print('Land Property')

class Father(GrandFather):
    def bank_balance(self):
        print('Big Bank Balance')

class Son(Father):
    def political_power(self):
        print('Political Power')

# Usage
s = Son()
s.land()           # From GrandFather
s.bank_balance()   # From Father
s.political_power() # Own method
```

Logic: Son inherits land from grandfather, money from father, and has own political power.

2.4 Hybrid Inheritance

Definition: Combination of two or more types of inheritance.

Example:

```
python
```

```
class Hybrid(Mobile, Father):  # Multiple + Multilevel
    def info(self):
        print('This is a Hybrid Class')

# Usage
h = Hybrid()
h.speaker()         # From JBL (via Mobile)
h.bank_balance()    # From Father
h.info()            # Own method
```

Logic: Combines features from different inheritance types.

🔄 3. Polymorphism

Definition: Polymorphism means "one name, many forms." Same function name behaves differently for different classes or data types.

Real-life Example: The word "run" - humans run, cars run, programs run - same word, different meanings.

3.1 Built-in Polymorphism

```python
# + operator works differently
print(5 + 5)         # 10 (addition)
print('5' + '5')     # '55' (concatenation)
print([1,2] + [3,4]) # [1,2,3,4] (list joining)
```

3.2 Method Overriding

Definition: Child class changes the behavior of parent's method.

Example:

```python
class Car:
    def speed(self):
        return '200 km/hr'

class Audi(Car):
    def speed(self):  # Override parent method
        return '300 km/hr'

class BMW(Car):
    def speed(self):  # Override parent method
        return '250 km/hr'

# Usage
c = Car()
a = Audi()
b = BMW()

print(c.speed())  # 200 km/hr
print(a.speed())  # 300 km/hr
print(b.speed())  # 250 km/hr
```

Logic: Same method name `speed()` but different behavior for each car.

3.3 Method Overloading (Python Way)

Definition: Same method name with different parameters.

Example:

```python
class Greet:
    def hello(self, name=None):
        if name:
            print(f'Hello {name}!')
        else:
            print('Hello!')

# Usage
g = Greet()
g.hello()          # Hello!
g.hello('Ankit')   # Hello Ankit!
```

Logic: Same method `hello()` works with or without parameters.

🎭 4. Abstraction

Definition: Abstraction hides complex implementation and shows only the relevant details. It's like designing an interface.

Real-life Example: TV Remote - you press power button, TV turns on. You don't need to know the internal circuit.

Python Implementation: Uses `abc` module (Abstract Base Class).

Example:

```python
```

```
from abc import ABC, abstractmethod

class Website(ABC):  # Abstract class

    @abstractmethod
    def login(self):
        pass  # Must be implemented by child

    @abstractmethod
    def logout(self):
        pass  # Must be implemented by child

class MySite(Website):
    def login(self):
        print('Login Done!')

    def logout(self):
        print('Logout Done!')

# Usage
s = MySite()
s.login()
s.logout()

# Note: You CANNOT create object of Website directly!
# w = Website()  # This will give error
```

Logic:

- Abstract class defines what methods must exist
- Child class decides how to implement them

- Forces consistent interface across all child classes

📚 5. Complete OOP Concepts Recap

Concept	Description	Example
Class	Blueprint for creating objects	`class Car:`
Object	Real instance created from class	`my_car = Car()`
Inheritance	Child acquires parent's properties	`class Audi(BMW):`
Polymorphism	Same function, different behavior	`speed()` method in different cars
Abstraction	Hide complex details, show interface	Abstract `Website` class
Encapsulation	Bundle data and methods, protect with access modifiers	Private `__balance` in bank account

---

🌟 6. Real-World Examples Summary

Banking System Example

```
python
```

```python
class Bank:  # Abstract parent
    def interest_rate(self):
        pass

class SBI(Bank):
    def interest_rate(self):
        return "7% per year"

class HDFC(Bank):
    def interest_rate(self):
        return "6.5% per year"

# Polymorphism in action
banks = [SBI(), HDFC()]
for bank in banks:
    print(bank.interest_rate())
```

Vehicle Hierarchy

```
python
```

```
class Vehicle:      # Base class
    def move(self):
        print("Moving...")

class Car(Vehicle):     # Single inheritance
    def move(self):     # Override
        print("Driving on road")

class Boat(Vehicle):    # Single inheritance
    def move(self):     # Override
        print("Sailing on water")

class AmphibiousCar(Car, Boat):  # Multiple inheritance
    def move(self):     # Override
        print("Can move on land and water")
```

🎯 Key Takeaways

✅ **Inheritance** enables code reuse and creates parent-child relationships

✅ **Single Inheritance:** One parent → one child

✅ **Multiple Inheritance:** Many parents → one child

✅ **Multilevel Inheritance:** Chain of inheritance (grandparent → parent → child)

✅ **Hybrid Inheritance:** Combination of multiple types

✅ **Polymorphism** allows same method name to behave differently

✅ **Method Overriding** changes parent's method behavior in child

✅ **Abstraction** hides complexity and provides clean interfaces

✅ **Abstract classes** cannot be instantiated directly

✅ **OOP makes code modular, reusable, and maintainable**

📝 Practice Questions

Question 1: Single Inheritance

Create a `Vehicle` class with `start()` method. Create a `Car` class that inherits from `Vehicle` and adds `honk()` method.

Answer:

```python
class Vehicle:
    def start(self):
        print("Vehicle started")

class Car(Vehicle):
    def honk(self):
        print("Beep beep!")

# Usage
my_car = Car()
my_car.start()  # Inherited method
my_car.honk()   # Own method
```

Question 2: Method Overriding

Create `Animal` class with `sound()` method. Create `Dog` and `Cat` classes that override the `sound()` method.

Answer:

```python
class Animal:
    def sound(self):
        print("Animal makes sound")

class Dog(Animal):
    def sound(self):
        print("Woof!")

class Cat(Animal):
    def sound(self):
        print("Meow!")

# Usage
animals = [Dog(), Cat()]
for animal in animals:
    animal.sound()  # Polymorphism in action
```

Question 3: Multiple Inheritance

Create `Flyable` and `Swimmable` classes. Create `Duck` class that inherits from both.

Answer:

```python
```

```python
class Flyable:
    def fly(self):
        print("Flying in the sky")

class Swimmable:
    def swim(self):
        print("Swimming in water")

class Duck(Flyable, Swimmable):
    def quack(self):
        print("Quack quack!")

# Usage
duck = Duck()
duck.fly()      # From Flyable
duck.swim()     # From Swimmable
duck.quack()    # Own method
```

Question 4: Abstraction

Create an abstract `Shape` class with `area()` method. Create `Circle` and `Rectangle` classes.

Answer:

```
python
```

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

# Usage
circle = Circle(5)
rectangle = Rectangle(4, 6)

print(f"Circle area: {circle.area()}")
print(f"Rectangle area: {rectangle.area()}")
```

🏆 Assignment Questions

Assignment 1: University System

Create a university system with:

- `Person` class (name, age)
- `Student` class inheriting from `Person` (student_id, course)
- `Teacher` class inheriting from `Person` (employee_id, subject)
- `TechStudent` class inheriting from `Student` (programming_language)

Assignment 2: Media Player

Create a media player system with:

- Abstract `MediaPlayer` class with `play()` and `stop()` methods
- `AudioPlayer` class implementing audio playback
- `VideoPlayer` class implementing video playback
- `UniversalPlayer` class inheriting from both (multiple inheritance)

Assignment 3: Banking System

Create a banking system with:

- `Bank` class with basic banking operations
- `SavingsAccount` class inheriting from `Bank`
- `CurrentAccount` class inheriting from `Bank`
- Override `interest_rate()` method in each account type
- Demonstrate polymorphism with different account types

Assignment 4: Gaming Characters

Create a game character system with:

- `Character` class (name, health, attack)
- `Warrior` class inheriting from `Character` (sword_skill)
- `Mage` class inheriting from `Character` (magic_power)
- `Archer` class inheriting from `Character` (bow_skill)
- Override `attack()` method in each character type

Assignment 5: Transportation System

Create a transportation system with:

- Abstract `Transport` class with `move()` method
- `LandTransport` class for land vehicles
- `WaterTransport` class for water vehicles
- `AirTransport` class for air vehicles
- `Amphibious` class inheriting from multiple transport types

💡 Remember

- **Inheritance** creates parent-child relationships
- **Polymorphism** allows same method name, different behavior
- **Abstraction** hides complexity, shows only necessary details
- **Use real-life examples** to understand concepts better
- **Practice with different inheritance types**
- **Abstract classes** cannot be instantiated directly
- **Method overriding** changes parent's method in child class

---

🚀 Next Steps

1. Practice all inheritance types with your own examples
2. Create abstract classes for common interfaces
3. Use polymorphism to make your code flexible
4. Combine all OOP concepts in larger projects
5. Study real-world applications of OOP patterns

Happy Learning! 🎉