Object-Oriented Programming and Inheritance

Chapter -3

Class and Object

- **What is a Class?**
- A **Class** is like a **blueprint, plan, or template**.
- It describes how something should look and what it can do.
- It does not hold data itself, but it tells how the data should be stored.
- Example in real life:
- **Blueprint of a House** → Defines number of rooms, doors, windows.
- But it is not an actual house

Create a Class

To create a class, use the keyword class:

Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
print(MyClass)
```

Create an object named p1, and print the value of x:

- Object
- **What is an Object?**
- An **Object** is a **real-world entity created from a class**.
- If a class is a blueprint, then an object is the **actual house built from the blueprint**.
- Each object has its own data.
- Example in real life:
- From the house blueprint, you can build **House 1, House 2, House 3**.
- Each house can have different colors, sizes, and owners.

```
class MyClass:
  x = 5
p1 = MyClass()
print(p1.x)
```

- **1. Attributes in Python**
- **Attributes** are variables that belong to an object (inside a class).
- They store information about the object.

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

# Creating object
s1 = Student(“Aarav", 21)

print(s1.name)  # Output: Aarav
print(s1.age)   # Output: 21
```

Here, name and age are **attributes** of the object s1.

- **2. Methods in Python**
- **Methods** are functions written inside a class.
- They define the behavior of objects (what an object can do).

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

    # Method
    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Creating object
s1 = Student(“Aarav", 21)
s1.display_info()   # Output: Name: Aarav, Age: 21
```

Here, display_info() is a **method**

- __init__ → saves data **inside the object**. (object remembers)
- **normal method** → just uses data **for that moment**. (object forgets)

```
class myclass:
    "hello div A"
    x=5

    print(x)

r=myclass()
print(r.__doc__)
```

- **Constructors in Python**

•**constructor** is a special method used to **initialize objects**.
•In Python, the constructor is always written as __init__.
•It is called automatically when an object is created.
•class Student:
- # Constructor
- def __init__(self, name, age):
- self.name = name
- self.age = age

•# Object creation automatically calls constructor
s1 = Student(“Abha", 21)
s2 = Student("Ravi", 22)

print(s1.name, s1.age) # Output: Abha 21
print(s2.name, s2.age) # Output: Ravi 22

Inheritance in OOP

Definition

- Inheritance is an **Object-Oriented Programming (OOP)** concept.
- It allows a **child class (derived class)** to reuse the properties and methods of a **parent class (base class)**.
- It shows the **“is-a” relationship** between classes.

Example: A Car is a Vehicle, so Car can inherit from Vehicle.

Types of Inheritance

- **1. Single Inheritance**
- **Definition:** One child class inherits from only one parent class.
- **Structure:** One parent → One child.
- **Theory:** Simplest form of inheritance. Used when a child class is directly related to a single parent class.
- Example:

```
class Animal:
    def speak(self):
        print("This is an animal")

class Dog(Animal):
    def bark(self):
        print("Dog barks")

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

2. Multilevel Inheritance

- **Definition:** A chain of inheritance where a class inherits from another derived class.
- **Structure:** Parent → Child → Grandchild.
- **Theory:** Useful when classes form a step-by-step hierarchy.

```
class Grandparent:
    def show1(self):
        print("I am Grandparent")
class Parent(Grandparent):
    def show2(self):
        print("I am Parent")
class Child(Parent):
    def show3(self):
        print("I am Child")
c = Child()
c.show1()
c.show2()
c.show3()
```

- **3. Multiple Inheritance**
- **Definition:** A child class inherits from two or more parent classes.
- **Structure:** Parent1 + Parent2 → Child.
- **Theory:** Child class can access features of all parent classes.

```
class Father:
    def skills(self):
        print("Can drive")

class Mother:
    def hobbies(self):
        print("Can cook")

class Child(Father, Mother):
    def talents(self):
        print("Can dance")

c = Child()
c.skills()
c.hobbies()
c.talents()
```

- **4. Hybrid Inheritance**
- **Definition:** A combination of two or more types of inheritance.
- **Structure:** Mix of single, multiple, and multilevel.
- **Theory:** Represents complex real-world relationships.

```
class A:
    def showA(self):
        print("Class A")
class B(A):
    def showB(self):
        print("Class B")
class C(A):
    def showC(self):
        print("Class C")
class D(B, C):
    def showD(self):
        print("Class D")
d = D()
d.showA()
d.showB()
d.showC()
d.showD()
```

- **5. Hierarchical Inheritance**
- **Definition:** One parent class is inherited by multiple child classes.
- **Structure:** One parent → Many children.
- **Theory:** Useful when several child classes share common behavior of a single parent.

```
class Vehicle:
    def wheels(self):
        print("Has wheels")
class Car(Vehicle):
    def four_wheeler(self):
        print("Car has 4 wheels")
class Bike(Vehicle):
    def two_wheeler(self):
        print("Bike has 2 wheels")
car = Car()
car.wheels()
car.four_wheeler()
bike = Bike()
bike.wheels()
bike.two_wheeler()
```

◇ 2. HAS-A Relationship (Composition)

- Definition: When one class contains an object of another class, it forms a HAS-A relationship.
- Example: A Car has a Engine → Car HAS-A Engine

```
# Class Engine
class Engine:
    def start(self):
        print("Engine starts")
# Class Car has an Engine (composition)
class Car:
    def __init__(self):
        self.engine = Engine()  # Car HAS-A Engine
```

🔑 Key Difference:

IS-A → Implemented using Inheritance (class Dog(Animal): ...)

HAS-A → Implemented using Composition (one class contains another as an object).

```
    def start(self):
        print("Car is starting...")
        self.engine.start()


# Usage
c = Car()
c.start()
```

- ◇ **What is Method Overriding?**
- **Definition**: When a **child class** defines a method with the **same name** as a **parent class method**, the child's method **replaces (overrides)** the parent's method when called through a child object.
- Used in **Inheritance** to change or extend parent behavior.

```
# Parent class
class Animal:
    def speak(self):
        print("Animal makes a sound")
# Child class overrides the method
class Dog(Animal):
    def speak(self):   # overriding speak() method
        print("Dog barks")
```

```
# Another child class
class Cat(Animal):
    def speak(self):   # overriding speak() method
        print("Cat meows")
# Usage
a = Animal()
a.speak()   # Output: Animal makes a sound
d = Dog()
d.speak()   # Output: Dog barks

c = Cat()
c.speak()   # Output: Cat meows
```

Here, Dog and Cat override the speak() method of Animal