



CHAPTER-3

OBJECT ORIENTED PROGRAMMING USING PYTHON

Introduction

In this chapter, students will learn the fundamentals of Object-Oriented Programming (OOP) using Python. They will explore key OOP concepts like classes and objects to model real-world entities in code. The chapter covers encapsulation, inheritance and polymorphism to create flexible, adaptable programs.

3.1 Object-Oriented Programming (OOP) Using Python

Q. What is Object-Oriented Programming (OOP)? Explain its benefits.

Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a method of designing programs using objects and classes. In Object-Oriented Programming, a program is developed by using objects and classes. Each object represents a real-world entity with specific properties and behaviors. A class acts as a template for creating many objects with similar characteristics. OOP uses encapsulation, inheritance, and polymorphism to manage data and behavior efficiently. This approach helps programmers model real-world problems in software development.

Benefits of OOP

- i. **Modularity**
OOP divides a program into small modules. This makes the program easier to understand, manage, and develop.
- ii. **Reusability**
Classes and objects can be reused in different programs or different parts of the same program. This reduces code repetition and saves development time.
- iii. **Easier Maintenance**
OOP programs are easier to maintain and update. Changes in one part of the program do not affect the entire system.
- iv. **Scalability and Adaptability**
The structure of OOP supports scalability and adaptability. New features can be added easily as the software grows.
- v. **Better Collaboration**
OOP makes the code more organized and easier to understand. This helps multiple programmers work together more effectively.



TIDBIT

- Object-Oriented Programming helps programmers think in terms of real-life objects, like students or cars, making programs easier to understand and manage.

CLASS ACTIVITY

Create a chart comparing procedural programming and OOP. Have students explain scenarios where OOP is more useful.

Sol:

Procedural Programming vs Object-Oriented Programming (OOP)

Procedural Programming	Object-Oriented Programming (OOP)
Programs are divided into functions.	Programs are divided into classes and objects.
It follows a step-by-step approach.	It follows a real-world object approach.
Code reuse is limited.	Code reuse is supported through inheritance and classes.
It is less suitable for large programs.	It is suitable for large and complex programs.
Maintenance becomes difficult as programs grow.	Programs are easier to maintain and update.
Data and functions are separate.	Data and methods are kept together in classes.
It provides less security for data.	It supports encapsulation for better data protection.
It is best for small and simple programs.	It is best for large and reusable software projects.

Scenarios Where OOP is More Useful

- School Management System**
OOP can manage students, teachers, and courses using classes and objects.
- Banking System**
OOP can manage accounts, transactions, and customer information securely.
- Online Shopping System**
OOP can handle products, customers, payments, and orders efficiently.
- Library Management System**
OOP can organize books, members, and borrowing records easily.
- Game Development**
OOP can represent players, enemies, and game objects as separate classes.
- Hospital Management System**
OOP can manage doctors, patients, appointments, and medical records effectively.

3.2 Core Concepts of OOP

Ex.Q1. Explain the concept of classes and objects in Object-Oriented Programming (OOP).

Core Concepts of OOP

The core concepts of Object-Oriented Programming (OOP) are classes and objects, encapsulation, inheritance, and polymorphism.

These concepts make programs well organized, reusable, and easier to manage. They help programmers write structured programs and model real-world problems more effectively.

Classes and Objects – Representing Real-World Entities in Code

- A class is a **blueprint or template** that defines the properties (attributes) and actions (methods) of an entity.
- An object is an **instance of a class** that contains actual values for its attributes.

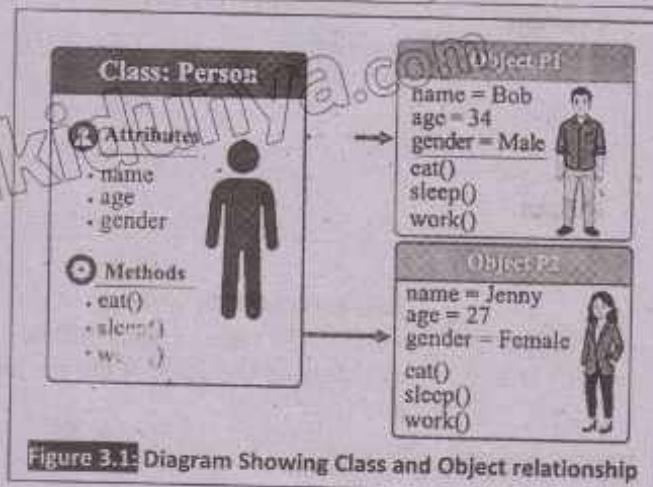


Figure 3.1: Diagram Showing Class and Object relationship

A class defines what an object will have and what it can do. An object is created from a class and represents a real-world entity. A class can be used to create many objects with similar characteristics. Using classes and objects keeps data and related functions together, making programs more organized and easier to understand.

Example

Write a Python program to create a Student class with a name attribute and a display() method. Create two student objects and display their names.

Program:

```
class Student:
    def __init__(self, name):
        self.name = name
    def display(self):
        print("Student Name:", self.name)
# Creating objects
object1 = Student("Ali")
object2 = Student("Ahmed")
# Using objects
object1.display()
object2.display()
```

Output:

```
Student Name: Ali
Student Name: Ahmed
```

Explanation:

- **class Student** → Defines a blueprint for student objects.
- **__init__()** → Sets the student's name when an object is created.
- **self.name** → Stores the student's name in the object.
- **display()** → Displays the student's name on the screen.
- **object1 and object2** → Two different objects of the Student class.
- Each object has its own data (Ali and Ahmed).

Q. What is the **__init__()** method and **self** in Python?

__init__() Method

The **__init__()** method is a **special method** in Python. It is **automatically called** when an object of a class is created. It is mainly used to initialize the object's data (attributes).

Example:

```
class Student:
    def __init__(self, age):
        self.age = age
s1 = Student(17)
print(s1.age)
```

Output: 17

Explanation:

- **__init__()** runs automatically when **Student(17)** is executed.
- It stores the value 17 in the object's age attribute.
- **print(s1.age)** displays the value of the age attribute.

self in Python

self is a reference to the current object of a class. It is used to access the object's attributes and methods inside the class.

Example:

```
class Student:
    def __init__(self, name):
        self.name = name
    def display(self):
        print(self.name)
s1 = Student("Ijaz")
s1.display()
```

Output: Ijaz

Explanation:

- self.name stores the name of the current object.
- self allows each object to keep its own data.

Q. What is the pass statement in Python and why it is used?

The pass statement is a **null statement** in Python. It does nothing when executed. It is used as a **placeholder** where a statement is required but no code is written yet.

Example:

```
class Student:
    pass
s1 = Student()
print("Object created successfully.")
```

Output: Object created successfully.

Why is pass used in OOP in Python?

In Object-Oriented Programming (OOP), pass is used to create **empty classes or methods** that will be completed later.

Example

```
class Student:
    pass
```

The Student class has no attributes or methods, but it is a valid class because of the pass statement.

Ex.Q.8 What are the key differences between a class and an object in Object-Oriented Programming?

Ans:

Difference Between a Class and an Object

Class	Object
A class is a blueprint or template used to create objects.	An object is a real instance that is created from a class.
A class defines the properties (attributes) and methods (functions) of an object.	An object contains actual values for the attributes and can use the methods of the class.
One class can be used to create many objects.	Each object is created from one class.
A class does not represent a real item. It only describes how objects will be created.	An object represents a real-world person, place, thing, or idea.
A class is created only once.	Many objects can be created from the same class.
Memory for data is not allocated when a class is created.	Memory for data is allocated when an object is created.
A class is created by using the class keyword in Python.	An object is created by calling the class name with parentheses, such as Student().
Example: class Student:	Example: student1 = Student()

ADDITIONAL PROGRAMS

1. Write a Python program to create a Rectangle class with a method to calculate and display its area.

Program:

```
class Rectangle:
    def area(self, length, width):
        print("Area =", length * width)
rect = Rectangle()
rect.area(5, 4)
```

Output: Area = 20

2. Write a Python program to create a Book class with a title attribute and a display() method. Create two book objects and display their titles.

Program:

```
class Book:
    def __init__(self, title):
        self.title = title
    def display(self):
        print("Book Title:", self.title)
book1 = Book("Python Programming")
book2 = Book("Computer Science")
book1.display()
book2.display()
```

Output:

Book Title: Python Programming
Book Title: Computer Science

3. Write a Python program to create an Employee class with name and salary attributes. Create two employee objects and display their information.

Program:

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
    def display(self):
        print("Name:", self.name)
        print("Salary:", self.salary)
emp1 = Employee("Ali", 50000)
emp2 = Employee("Ahmed", 60000)
emp1.display()
emp2.display()
```

Output:

Name: Ali
Salary: 50000
Name: Ahmed
Salary: 60000

Ex.Q.2: What is encapsulation in OOP? Why is it important?

Encapsulation

Encapsulation is a concept in Object-Oriented Programming (OOP) in which data (variables) and methods (functions) are combined into a single unit called a class. It also restricts direct access to the data. In encapsulation, data and the methods that work on that data are kept together inside the same class. This helps protect the data from unwanted access. Unlike Java and C++, Python does not use strict private or protected keywords. Instead, it uses naming conventions to show how class members should be accessed.

Why Encapsulation is Important

Protects Data from Unwanted Access

Encapsulation prevents direct access to important data and helps keep it safe.

Improves Security

It protects data from unauthorized changes and improves the security of the program.

Makes Code Easier to Manage

Keeping data and methods together makes the program more organized and easier to understand.

Allows Controlled Modification

Data can only be changed through specific methods, which helps control how it is modified.

Example

A bank account is a good example of encapsulation. The account balance is hidden from direct access. A user can change the balance only by using methods such as `deposit()` and `withdraw()`. The balance cannot be changed directly.

Levels of Access Control

Python provides different levels of access control by using naming prefixes.

Public Members

Public members can be accessed from anywhere, inside or outside the class. By default, all members in Python are public.

Syntax: `self.attribute_name`

Protected Members

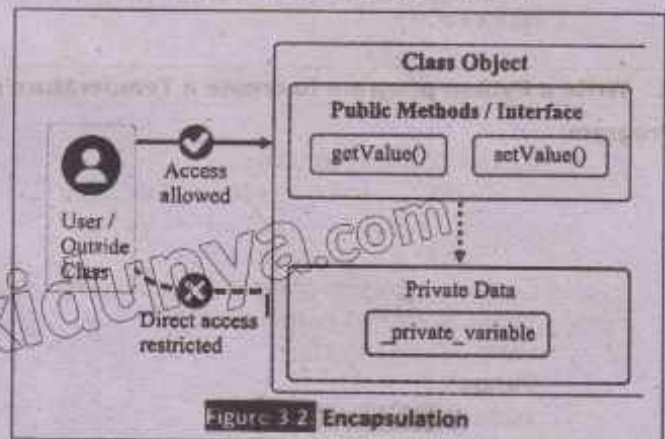
Protected members are meant to be used only within the class and its subclasses. This is only a naming convention and does not strictly prevent access from outside the class.

Syntax: `self._attribute_name` (single underscore)

Private Members

Private members are intended to be accessed only inside the class. Python uses a technique called Name Mangling to reduce accidental access from outside the class.

Syntax: `self.__attribute_name` (double underscore)



Example:

```
class BankAccount:
    def __init__(self, balance):
        self._balance = balance # Private variable
    def deposit(self, amount):
        self._balance += amount
    def withdraw(self, amount):
        if amount <= self._balance:
            self._balance -= amount
        else:
            print("Insufficient balance")
    def get_balance(self):
        return self._balance

# Creating an object
account = BankAccount(1000)
account.deposit(500)
account.withdraw(300)
print(account.get_balance()) # Output: 1200
# Direct access is not allowed
# print(account._balance) # Error
Output: 1200
```

ADDITIONAL PROGRAMS

1. Write a Python program to create a Marks class that stores and displays a student's marks.

Program:

```
class Marks:
    def __init__(self, marks):
        self.__marks = marks
    def show_marks(self):
        print("Marks:", self.__marks)
student = Marks(85)
student.show_marks()
Output:
Marks: 85
```

2. Write a Python program to create a Temperature class that stores and displays temperature.

Program:

```
class Temperature:
    def __init__(self, value):
        self.__value = value
    def display(self):
        print("Temperature:", self.__value, "°C")
temp = Temperature(30)
temp.display()
Output:
Temperature: 30 °C
```

3. Write a Python program to create a Product class that stores and displays the price of a product.

Program:

```
class Product:
    def __init__(self, price):
        self.__price = price
    def display_price(self):
        print("Price:", self.__price)
item = Product(500)
item.display_price()
Output: Price: 500
```

4. Write a Python program to create an Employee class that manages an employee's salary.

Program:

```
class Employee:
    def __init__(self, salary):
        self.__salary = salary
    def increase_salary(self, amount):
        self.__salary += amount
    def decrease_salary(self, amount):
        self.__salary -= amount
    def get_salary(self):
        return self.__salary
emp = Employee(50000)
emp.increase_salary(5000)
emp.decrease_salary(2000)
print(emp.get_salary()) # Output: 53000
Output: 53000
```

5. Write a Python program to create a MobileBalance class that manages a mobile account balance.

Program:

```
class MobileBalance:
    def __init__(self, balance):
        self.__balance = balance
    def recharge(self, amount):
        self.__balance += amount
    def use_balance(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient balance")
    def get_balance(self):
        return self.__balance
mobile = MobileBalance(200)
mobile.recharge(100)
mobile.use_balance(50)
print(mobile.get_balance())
Output: 250
```

6. Write a Python program to create a LibraryBook class that manages the number of available books.

Program:

```
class LibraryBook:
    def __init__(self, copies):
        self.__copies = copies
    def add_books(self, number):
        self.__copies += number
    def issue_book(self, number):
        if number <= self.__copies:
            self.__copies -= number
        else:
            print("Books are not available.")
    def get_copies(self):
        return self.__copies
book = LibraryBook(20)
book.add_books(5)
book.issue_book(3)
print(book.get_copies()) # Output: 22
Output: 22
```



Additional Multiple Choice Questions

- The term OOP stands for:
A) Object Operation Program
B) Object-Oriented Programming
C) Online Object Processing
D) Operating Object Program
- A method of designing programs using objects and classes is called:
A) Object-Oriented Programming
B) Structured Programming
C) Modular Programming
D) Event Programming
- A template used to create many similar objects is called:
A) Method
B) Object
C) Class
D) Variable
- Encapsulation is the concept in OOP in which data and methods are combined into a single unit called:
A) Object
B) Class
C) Function
D) Module

5. Which member can be accessed from anywhere by default in Python?
A) Private member B) Protected member C) Hidden member D) Public member
6. A preliminary design used to define properties and methods of an entity is called:
A) Object B) Instance C) Class D) Function
7. Which naming style represents a private member in Python?
A) self.name B) self_name C) self.__name D) self:name
8. Why is encapsulation important in OOP?
A) It increases screen brightness. B) It protects data from unwanted access.
C) It removes all errors automatically. D) It speeds up internet connection.

Answers:

1	B	2	A	3	C	4	B	5	D	6	C	7	C	8	B
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Ex.Q.3 What is inheritance? Explain its types.

Inheritance

Inheritance is a feature of Object-Oriented Programming (OOP) that allows one class to take the properties and methods of another class. Inheritance allows a new class to reuse the features of an existing class. The new class is called the subclass, while the existing class is called the parent class or superclass. The subclass can also add new features or modify the existing ones. Inheritance promotes code reuse, reduces repetition, and makes programs easier to extend without rewriting existing code.

Types of Inheritance

1. Single Inheritance

Single inheritance occurs when one child class inherits attributes and methods from only one parent class.

In single inheritance, the child class reuses the features of one parent class and can also add new features. This is the simplest and most common type of inheritance in Python.

- **Parent Class (Base Class)**

A parent class also called a base class or **superclass** is the class whose properties and methods are inherited by another class.

- **Child Class (Derived Class)**

A child class also called a derived class or **subclass** is the class that inherits properties and methods from the parent class. It is created by writing the parent class name inside parentheses.

Syntax: `class Child(Parent):`

- **Method Overriding**

Method overriding allows a child class to provide its own version of a method that already exists in the parent class. The `super()` function can be used to call a method or constructor of the parent class.

Example

If Animal is the parent class, then Dog can be the child class. The Dog class inherits the features of the Animal class and can also define its own methods.

Short Question

What is single inheritance in OOP?

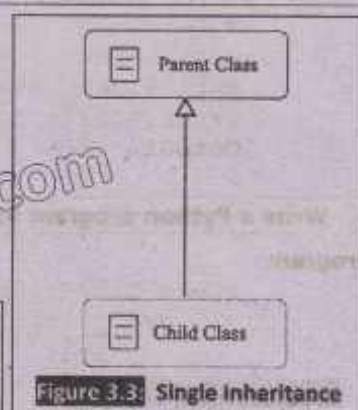


Figure 3.3: Single Inheritance

Short Question

What is Method overriding in OOP?

Example 1: Write a Python program to demonstrate single inheritance using Animal and Dog classes.

```
# Parent Class (Superclass)
class Animal:
    def __init__(self, name):
        self.name = name
    def eat(self):
        print(f"{self.name} is eating.")

# Child Class (Subclass)
class Dog(Animal):
    def bark(self):
        print(f"{self.name} says Woof!")

# Creating an object of the child class
my_dog = Dog("Buddy")
# Calling methods
my_dog.eat()
my_dog.bark()
```

Output:

```
Buddy is eating.
Buddy says Woof!
```

2. Multiple Inheritance

Multiple inheritance occurs when one child class inherits properties and methods from more than one parent class. This allows a child class to combine features from different parent classes.

Example

A Smartphone class can inherit features from both a Camera class and a Computer class.

Short Question

What is Multiple inheritance in OOP?

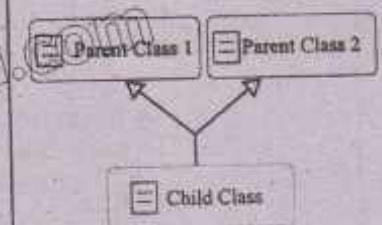


Figure 3.4: Multiple Inheritance

Example 2:

Write a Python program to demonstrate multiple inheritance using Father, Mother, and Child classes.

Program:

```
# Parent Class 1
class Father:
    def skill1(self):
        print("Father's skill: Gardening")

# Parent Class 2
class Mother:
    def skill2(self):
        print("Mother's skill: Cooking")

# Child Class inheriting from both parent classes
class Child(Father, Mother):
    def own_skill(self):
        print("Child's skill: Coding")

# Creating an object of the Child class
c = Child()
# Calling methods
c.skill1()
c.skill2()
c.own_skill()
```

Output:

```
Father's skill: Gardening
Mother's skill: Cooking
Child's skill: Coding
```

3. Multilevel Inheritance

Multilevel inheritance occurs when one child class inherits from another child class (derived class), creating a chain of inheritance.

In this type of inheritance, the last child class can use the features of both its parent class (immediate parent) and its grandparent class.

Example

If Animal is the parent class, Mammal inherits from Animal, and Dog inherits from Mammal, then Dog can use the features of both Animal and Mammal.

Short Question
What is Multilevel Inheritance in OOP?

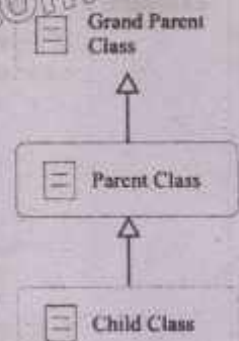


Figure 3.5: Multilevel Inheritance

Example:

Write a Python program to demonstrate multilevel inheritance using Vehicle, Car, and ElectricCar classes.

Program:

```
# Grandparent Class
class Vehicle:
    def start_engine(self):
        print("Engine started.")

# Parent Class
class Car(Vehicle):
    def drive(self):
        print("Car is driving.")

# Child Class
class ElectricCar(Car):
    def charge(self):
        print("Battery is charging.")

# Creating an object of the child class
my_tesla = ElectricCar()

# Calling methods
my_tesla.start_engine()
my_tesla.drive()
my_tesla.charge()

Output:
Engine started.
Car is driving.
Battery is charging.
```

? DO YOU KNOW?

- A class is like a blueprint, while an object is the real thing created from that blueprint.

TIDBIT

- Self is the reference to the current object, passed automatically to instance methods.



CLASS ACTIVITY

Create a Vehicle class and extend it with Car and Bike subclasses. Add unique methods to each subclass and demonstrate inheritance.

Program:

```
# Parent Class
class Vehicle:
    def start(self):
        print("Vehicle started.")
# Child Class 1
class Car(Vehicle):
    def drive(self):
        print("Car is driving.")
# Child Class 2
class Bike(Vehicle):
    def ride(self):
        print("Bike is riding.")
my_car = Car()
my_bike = Bike()
my_car.start()
my_car.drive()
my_bike.start()
my_bike.ride()
```

Output:

```
Vehicle started.
Car is driving.
Vehicle started.
Bike is riding.
```

Q. What is the super() function in Python?

The super() function is a built-in Python function used to access the methods and constructor of the parent class from the child class. It helps reuse the parent class code and avoids code duplication.

Example:

```
class Animal:
    def __init__(self, name):
        self.name = name
    def sound(self):
        print("Animals make sounds.")
class Dog(Animal):
    def __init__(self, name):
        super().__init__(name)
    def display(self):
        print("Dog Name:", self.name)
d1 = Dog("Tommy")
d1.sound()
d1.display()
```

Output

```
Animals make sounds.
Dog Name: Tommy
```

ADDITIONAL PROGRAMS

Single Inheritance Programs

1. Write a Python program to demonstrate single inheritance using Person and Student classes.

Program:

```
class Person:
    def show_name(self):
        print("Name: Ali")
class Student(Person):
    def show_class(self):
        print("Class: 12th")
std = Student()
std.show_name()
std.show_class()
```

Output:

Name: Ali
Class: 12th

2. Write a Python program to demonstrate inheritance using Animal and Bird classes.

Program:

```
class Animal:
    def eat(self):
        print("Animals eat food.")
class Bird(Animal):
    def fly(self):
        print("Birds can fly.")
b = Bird()
b.eat()
b.fly()
```

Output:

Animals eat food.
Birds can fly.

3. Write a Python program to demonstrate single inheritance using Vehicle and Car classes.

Program:

```
class Vehicle:
    def start(self):
        print("Vehicle is starting.")
class Car(Vehicle):
    def drive(self):
        print("Car is moving.")
c1 = Car()
c1.start() # Inherited method
c1.drive() # Car class method
```

Output:

Vehicle is starting.
Car is moving.

Multiple Inheritance Programs

4. Write a Python program to demonstrate multiple inheritance using Teacher, Coach, and Student classes.

Program:

```
class Teacher:
    def teach(self):
        print("Teacher teaches Mathematics.")
class Coach:
    def train(self):
        print("Coach trains students.")
class Student(Teacher, Coach):
    def study(self):
        print("Student studies Computer Science.")
s = Student()
s.teach()
s.train()
s.study()
```

Output:

Teacher teaches Mathematics.
Coach trains students.
Student studies Computer Science.

5. Write a Python program to demonstrate multiple inheritance using Printer, Scanner, and AllInOne classes.

Program:

```
class Printer:
    def print_document(self):
        print("Printing document.")
class Scanner:
    def scan_document(self):
        print("Scanning document.")
class AllInOne(Printer, Scanner):
    def copy_document(self):
        print("Copying document.")
device = AllInOne()
device.print_document()
device.scan_document()
device.copy_document()
```

Output:

Printing document.
Scanning document.
Copying document.

6. Write a Python program to demonstrate multiple inheritance using Writer, Speaker, and Student classes.

Program:

```
class Writer:
    def write(self):
        print("Student can write essays.")
class Speaker:
    def speak(self):
        print("Student can deliver speeches.")
class Student(Writer, Speaker):
    def study(self):
```

```

        print("Student is studying.")
s1 = Student()
s1.write()    # Inherited from Writer
s1.speak()    # Inherited from Speaker
s1.study()    # Student class method

```

Output

Student can write essays.
 Student can deliver speeches.
 Student is studying.

Multilevel Inheritance Programs

7. Write a Python program to demonstrate multilevel inheritance using GrandFather, Father, and Son.

Program:

```

class GrandFather:
    def house(self):
        print("GrandFather owns a house.")
class Father(GrandFather):
    def car(self):
        print("Father owns a car.")
class Son(Father):
    def bike(self):
        print("Son owns a bike.")
s = Son()
s.house()
s.car()
s.bike()

```

Output:

GrandFather owns a house.
 Father owns a car.
 Son owns a bike.

8. Write a Python program to demonstrate multilevel inheritance using Vehicle, Car, and SportsCar classes.

Program:

```

class Vehicle:
    def start(self):
        print("Vehicle started.")
class Car(Vehicle):
    def drive(self):
        print("Car is moving.")
class SportsCar(Car):
    def speed(self):
        print("Sports car runs very fast.")
my_car = SportsCar()
my_car.start()
my_car.drive()
my_car.speed()

```

Output:

Vehicle started.
 Car is moving.
 Sports car runs very fast.

9. Write a Python program to demonstrate multilevel inheritance using Person, Teacher, and Principal classes.

Program:

```
class Person:
    def speak(self):
        print("Person can speak.")
class Teacher(Person):
    def teach(self):
        print("Teacher teaches students.")
class Principal(Teacher):
    def manage(self):
        print("Principal manages the school.")
p = Principal()
p.speak()
p.teach()
p.manage()
```

Output:

Person can speak.
Teacher teaches students.
Principal manages the school.

10. Write a Python program that create an Animal class and extend it with Dog and Cat subclasses. Add unique methods to each subclass and demonstrate inheritance.

Program:

```
class Animal:
    def eat(self):
        print("Animals eat food.")
class Dog(Animal):
    def bark(self):
        print("Dog barks.")
class Cat(Animal):
    def meow(self):
        print("Cat meows.")
my_dog = Dog()
my_cat = Cat()
my_dog.eat()
my_dog.bark()
my_cat.eat()
my_cat.meow()
```

Output:

Animals eat food.
Dog barks.
Animals eat food.
Cat meows.

11. Write a Python program that create an Employee class and extend it with Manager and Programmer subclasses. Add unique methods to each subclass and demonstrate inheritance.

Program:

```
class Employee:
    def work(self):
        print("Employees perform their duties.")
class Manager(Employee):
    def manage(self):
        print("Manager manages the team.")
```

```

class Programmer(Employee):
    def code(self):
        print("Programmer writes code.")
mgr = Manager()
prog = Programmer()
mgr.work()
mgr.manage()
prog.work()
prog.code()

```

Output:

Employees perform their duties.
 Manager manages the team.
 Employees perform their duties.
 Programmer writes code.

12. Write a Python program to demonstrate multilevel inheritance using University, Department, and Student classes, where each class has its own method.

Program:

```

class University:
    def university_info(self):
        print("University of Sargodha")
class Department(University):
    def department_info(self):
        print("Computer Science Department")
class Student(Department):
    def student_info(self):
        print("I am a Computer Science student")
# Create an object of Student class
s = Student()
# Calling methods
s.university_info()
s.department_info()
s.student_info()

```

Output

University of Sargodha
 Computer Science Department
 I am a Computer Science student



Additional Multiple Choice Questions

- A feature of OOP that allows one class to take the properties and methods of another class is called:
 A) Polymorphism B) Encapsulation C) Inheritance D) Abstraction
- In Object-Oriented Programming (OOP), inheritance allows a class to:
 A) Delete data B) Take properties and methods of another class
 C) Create variables only D) Stop program execution
- The existing class whose features are inherited is called the:
 A) Child class B) Parent class C) Object class D) Instance class
- When one child class inherits from only one parent class, it is called:
 A) Multiple inheritance B) Multilevel inheritance
 C) Single inheritance D) Hybrid inheritance
- A class that inherits properties and methods from another class is called:
 A) Parent class B) Superclass C) Derived class D) Blueprint

6. Which syntax is used to create a child class in Python?
A) class Child[Parent] B) class Child:Parent C) class Child(Parent) D) class Child(Parent)
7. Providing a new version of a parent class method in the child class is called:
A) Method calling B) Method overloading C) Method overriding D) Method hiding
8. A class that inherits features from two or more parent classes uses:
A) Single inheritance B) Multiple inheritance C) Multilevel inheritance D) Encapsulation
9. In multilevel inheritance, the last child class can use the features of:
A) Parent only B) Grandparent only C) Parent and grandparent D) Child only
10. Which function is used to call the parent class method in method overriding?
A) self() B) class() C) object() D) super()

Answers:

1	C	2	B	3	B	4	C	5	C	6	D	7	C	8	B	9	C	10	D
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----	---

Ex.Q.5 What is polymorphism in OOP, and how does it enhance flexibility in code?

Polymorphism

Polymorphism means "**many forms**". It is the ability of the same method, function, or operator to perform different actions depending on the object or situation.

In Object-Oriented Programming (OOP), polymorphism allows the same method or function to behave **differently for different objects**. This makes programs more flexible and reusable because the same method name can be used in different ways.

Example

In an online payment system, different payment methods such as Credit Card, UPI, NetBanking, and Wallet all use the same method named `processPayment()`. However, each payment method performs a different action. For example,

- NetBanking redirects the user to the bank login page.

The method name remains the same, but its behavior changes according to the payment type.

Types of Polymorphism

Polymorphism mainly has two types: compile-time polymorphism and runtime polymorphism.

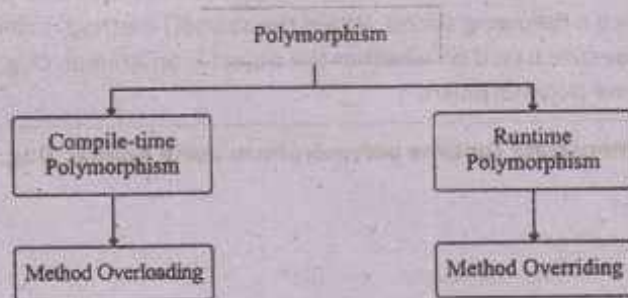


Figure 3.6: Polymorphism example showing the same function behaving differently

1. Compile-time Polymorphism

Compile-time polymorphism means the method or operation is selected during program compilation. It is usually achieved through method overloading or operator overloading in languages such as Java and C++.

Python does not support true method overloading based on function signatures. Since Python is dynamically typed, one function can accept different types of arguments. Similar behavior can be achieved by using default arguments, variable-length arguments (`*args`, `**kwargs`), or conditional statements inside the function.

Example

A `multiply()` method can work with different numbers of arguments by using default values or `*args`. This provides behavior similar to compile-time polymorphism.

Short Question

What is Compile-time polymorphism in OOP?

Write a Python program to demonstrate compile-time polymorphism using a Calculator class and a multiply() method.

Program:

```
class Calculator:
    def multiply(self, a=1, b=1, *args):
        result = a * b
        for num in args:
            result *= num
        return result
# Creating an object
calc = Calculator()
# Using default arguments
print(calc.multiply())
print(calc.multiply(4))
# Using multiple arguments
print(calc.multiply(2, 3))
print(calc.multiply(2, 3, 4))
```

Output:

```
1
4
6
24
```

2. Runtime Polymorphism (Method Overriding)

Runtime polymorphism means the method is selected while the program is running. It is achieved through method overriding.

In method overriding, a child class provides its own version of a method that is already defined in the parent class. The method that is executed depends on the object that calls it during program execution.

Example

Suppose the Animal class has a method named sound(). The Dog class overrides it to produce a barking sound, while the Cat class overrides it to produce a meowing sound. When the sound() method is called, Python decides at runtime which version of the method to execute based on whether the object is an Animal, Dog, or Cat. This dynamic behavior makes Python powerful for runtime polymorphism.

Short Question

What is Runtime polymorphism in OOP?

Write a Python program to demonstrate runtime polymorphism using Animal, Dog, and Cat classes.

Program:

```
class Animal:
    def sound(self):
        return "Some generic sound"
class Dog(Animal):
    def sound(self):
        return "Bark"
class Cat(Animal):
    def sound(self):
        return "Meow"
# Creating a list of objects
animals = [Dog(), Cat(), Animal()]
# Calling the same method for different objects
for animal in animals:
    print(animal.sound())
```

Output:

```
Bark
Meow
Some generic sound
```

Q. What are *args and **kwargs in Python?

*args

*args allows a function to accept **any number of positional arguments**. The values are stored as a **tuple**.

- * is a special Python syntax that collects multiple positional arguments.
- args is just a variable name. You can use another name, but args is the standard convention.

Example

```
def add(*args):  
    print(args)  
add(10, 20, 30)
```

Output:

```
(10, 20, 30)
```

**kwargs

kwargs allows a function to accept **any number of keyword arguments. The values are stored as a **dictionary**.

- ** is special Python syntax that collects multiple keyword arguments.
- kwargs is just a variable name. You can use another name, but kwargs is the standard convention.

Example

```
def display(**kwargs):  
    print(kwargs)  
display(name="Ali", age=18)
```

Output:

```
{'name': 'Ali', 'age': 18}
```

ADDITIONAL PROGRAMS

Compile-Time Polymorphism Programs

1. Write a Python program to demonstrate compile-time polymorphism using an add() method.

Program:

```
class Calculator:  
    def add(self, a=0, b=0, c=0):  
        return a + b + c  
calc = Calculator()  
print(calc.add())  
print(calc.add(5))  
print(calc.add(5, 10))  
print(calc.add(5, 10, 15))
```

Output:

```
0  
5  
15  
30
```

2. Write a Python program to demonstrate compile-time polymorphism using a display() method.

Program:

```
class Student:  
    def display(self, name="Ali", age=18):  
        print("Name:", name)  
        print("Age:", age)  
s = Student()  
s.display()  
s.display("Ahmed")
```

```
s.display("Sara", 20)
```

Output:

Name: Ali

Age: 18

Name: Ahmed

Age: 18

Name: Sara

Age: 20

3. Write a Python program to demonstrate compile-time polymorphism using an area() method.

Program:

```
class Rectangle:
    def area(self, length=1, width=1):
        return length * width

r = Rectangle()
print(r.area())
print(r.area(5))
print(r.area(5, 4))
```

Output

1

5

20

Runtime Polymorphism (Method Overriding) Programs

4. Write a Python program to demonstrate runtime polymorphism using Shape and Circle classes.

Program:

```
class Shape:
    def area(self):
        print("Area of a shape.")

class Circle(Shape):
    def area(self):
        print("Area of a circle.")
```

```
s = Shape()
```

```
c = Circle()
```

```
s.area()
```

```
c.area()
```

Output:

Area of a shape.

Area of a circle.

5. Write a Python program to demonstrate runtime polymorphism using Vehicle, Car, and Bike classes.

Program:

```
class Vehicle:
    def sound(self):
        return "Vehicle sound"

class Car(Vehicle):
    def sound(self):
        return "Car horn"

class Bike(Vehicle):
    def sound(self):
        return "Bike horn"

vehicles = {Car(), Bike(), Vehicle()}
```

```
vehicles:
    file.sound()
```

Car horn
Bike horn
Vehicle sound

Program:

Output
Manager is managing the team.
Developer is writing code.
Employee is working.



Additional Multiple Choice Questions

- A method providing different behavior for different objects is called:**

A) Inheritance B) Class C) Polymorphism D) Encapsulation
- In OOP, polymorphism allows the same method to:**

A) Delete objects B) Create classes only
C) Behave differently for different objects D) Stop program execution
- Which type of polymorphism selects the method during program compilation?**

A) Runtime polymorphism B) Compile-time polymorphism
C) Multiple inheritance D) Encapsulation
- In Python, behavior similar to method overloading can be achieved by using:**

A) Default arguments B) Parent class C) Private members D) Inheritance only
- Runtime polymorphism is achieved through:**

A) Encapsulation B) Method overriding C) Multiple inheritance D) Object creation
- The method executed during runtime depends on:**

A) Program size B) Number of variables C) Object that calls it D) Operating system
- Which statement best describes runtime polymorphism?**

A) Method is selected during compilation. B) Parent class cannot be used.
C) Method is selected while the program is running. D) Every class must have the same data.

Answers:

1	C	2	C	3	B	4	A	5	B	6	C	7	C
---	---	---	---	---	---	---	---	---	---	---	---	---	---

TEXTBOOK EXERCISE (SOLVED)



MULTIPLE CHOICE QUESTIONS

1. The OOP concept that hides data inside a class is:
A) Inheritance B) Polymorphism C) Encapsulation D) Abstraction
2. Classes and objects in Object-Oriented Programming are best described as:
A) Classes are real-world entities, and objects are methods.
B) Classes are blueprints, and objects are instances of those blueprints.
C) Classes are used for data storage, and objects are used for functionality.
D) Objects define functionality, and classes store data.
3. The main idea behind encapsulation in OOP is:
A) To hide data inside a class and prevent unauthorized access.
B) To create a flexible and adaptable program.
C) To reuse code from other classes.
D) To allow multiple inheritance in classes.
4. Inheritance in OOP means:
A) Reusing and extending functionality from a parent class to a child class.
B) Hiding data in a class.
C) Creating multiple instances of the same class.
D) Managing polymorphic behavior of classes.
5. Polymorphism in OOP is best defined as:
A) Allowing an object to take multiple forms through method overriding.
B) Hiding data inside a class for security.
C) Using multiple inheritance to extend classes. D) Ensuring that each object has only one method.
6. An example of inheritance in Python is:
A) Creating two classes with similar attributes.
B) A child class inheriting attributes and methods from a parent class.
C) Using private variables within a class. D) Modifying a method inside the same class.
7. The purpose of polymorphism in programming is:
A) To create reusable functions with multiple parameter types.
B) To allow a function or method to have different implementations.
C) To combine different classes into a single class. D) To manage the flow of control in a program.
8. The keyword used to define a class in Python is:
A) def B) class C) function D) object
9. Encapsulation is best described as:
A) Combining data and methods into a single unit and controlling access to data.
B) Combining multiple classes into one class.
C) Inheriting methods from other classes.
D) Allowing objects to have multiple forms.
10. Inheritance allows a class to:
(a) Hide its data only.
(b) Inherit attributes and methods from another class.
(c) Define multiple functions only.
(d) Create objects without a class.

Answers:

1	c	2	b	3	a	4	a	5	a	6	b	7	b	8	b	9	a	10	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----	---



SHORT QUESTIONS

1. **What is Object-Oriented Programming (OOP)?**

Ans: Object-Oriented Programming (OOP) is a method of designing programs using objects and classes. In Object-Oriented Programming, a program is developed by using objects and classes. Each object represents a real-world entity with specific properties and behaviors. A class acts as a template for creating many objects with similar characteristics.

2. **What is the purpose of a class in Python?**

Ans: A class is a blueprint or template that defines the properties (attributes) and actions (methods) of an entity. An object is an instance of a class that contains actual values for its attributes. A class defines what an object will have and what it can do.

3. **What is the difference between a class and an object?**

Ans:

Class	Object
A class is a blueprint or template used to create objects.	An object is a real instance that is created from a class.
A class defines the properties (attributes) and methods (functions) of an object.	An object contains actual values for the attributes and can use the methods of the class.
One class can be used to create many objects.	Each object is created from one class.
Example: <code>class Student:</code>	Example: <code>student1 = Student()</code>

4. **What is the purpose of `__init__()`?**

Ans: The `__init__()` method is a special method in Python that is called automatically when an object is created. Its purpose is to initialize the object's attributes with the required values.

5. **What is encapsulation in OOP, and why is it important?**

Ans: Encapsulation is a concept in Object-Oriented Programming (OOP) in which data (variables) and methods (functions) are combined into a single unit called a class. Encapsulation is important because it protects data from unauthorized or unwanted access, improves data security, and makes programs easier to manage and maintain.

6. **How does encapsulation help in keeping data secure and private?**

Ans: Encapsulation helps keep data secure and private by restricting direct access to an object's data. It allows the data to be accessed or modified only through defined methods. This protects the data from unauthorized or unwanted changes and helps maintain the correctness and security of the program.

7. **What is inheritance in OOP, and how does it help in reusing functionality?**

Ans: Inheritance is a feature of Object-Oriented Programming (OOP) that allows one class to take the properties and methods of another class. Inheritance helps in reusing functionality because a new class can use the existing features of another class without writing the same code again. This saves time, reduces code duplication, and makes programs easier to maintain.

8. **How does inheritance enable the extension of functionality in Python?**

Ans: Inheritance enables the extension of functionality in Python by allowing a new class to inherit the properties and methods of an existing class. The new class can also add new properties and methods or modify existing ones. This allows programmers to extend the functionality of a class without changing the original class.

9. **What is polymorphism in OOP, and how does it make code more flexible?**

Ans: Polymorphism means "many forms". It is the ability of the same method, function, or operator to perform different actions depending on the object or situation. This makes programs more flexible and reusable because the same method name can be used in different ways. This reduces code duplication and improves reusability.

10. How does polymorphism help in creating adaptable and reusable code?

Ans: Polymorphism helps in creating adaptable and reusable code by allowing the same method or function to work with different objects. Each object can perform its own action using the same method name. This reduces code duplication, improves code reusability, and makes programs easier to modify and maintain.



LONG QUESTIONS

1. Explain the concept of classes and objects in Object-Oriented Programming (OOP).

Ans: Solved in Chapter

2. What is encapsulation in OOP?

Ans: Solved in Chapter

3. How does inheritance work in OOP?

Ans: Inheritance

Inheritance is a feature of Object-Oriented Programming (OOP) that allows one class to take the properties and methods of another class. Inheritance allows a new class to reuse the features of an existing class.

How Inheritance Works

1. Create a Parent Class

- First, create a **parent class** that contains common properties (attributes) and methods (functions)
- These are the features that can be shared with other classes.

Example: An **Animal** class has a **name** property and an **eat()** method.

2. Create a Child Class

- Next, create a **child class** by inheriting the parent class.
- The child class is written by placing the parent class name inside parentheses.

Example: `class Dog(Animal):`

- **Dog** is the child class, and **Animal** is the parent class.

3. Use Parent Class Features

- The child class automatically gets all the properties and methods of the parent class.
- There is no need to write the same code again.

Example: If the **Animal** class has an **eat()** method, then the **Dog** class can also use **eat()**.

4. Add New Features

- The child class can have its own new properties and methods.
- These features are available only in the child class.

Example: The **Dog** class adds a **bark()** method that is not present in the **Animal** class.

5. Change Existing Features (Method Overriding)

- If needed, the child class can replace a method of the parent class with its own version. This is called **method overriding**.
- It allows the child class to perform the same task in a different way.

Example:

- The **Animal** class has a **sound()** method.
- The **Dog** class overrides it to return "**Woof!**" instead of the parent class output.

Write a Python program to demonstrate runtime polymorphism using Animal, Dog, and Cat classes.
Example:

```
class Animal:
    def eat(self):
        print("Animal can eat")
class Dog(Animal):
    def bark(self):
        print("Dog can bark")
dog1 = Dog()
dog1.eat()
dog1.bark()
Output:
Animal can eat
Dog can bark
```

4. What are the advantages of using inheritance in Python?

Ans: Advantages of Inheritance in Python

Inheritance is an OOP feature that allows a child class to use the properties and methods of a parent class. It helps make programs simple, organized, and easy to manage.

i. Code Reusability

- Inheritance allows a child class to reuse the code of the parent class.
- There is no need to write the same code again.
- This saves time and effort.

ii. Reduces Code Repetition

- Common properties and methods are written only once.
- Different child classes can use the same code.
- This keeps the program short and clean.

iii. Easy to Extend Programs

- A child class can add new properties and methods.
- It can also modify existing methods when needed.
- New features can be added without changing the parent class.

iv. Easy Maintenance

- Changes made in the parent class are available to child classes.
- This makes programs easier to update and maintain.
- It also reduces errors caused by repeated code.

v. Better Program Organization

- Common features stay in the parent class.
- Special features stay in the child class.
- This makes the code easy to understand and manage.

vi. Supports Method Overriding

- A child class can provide its own version of a parent class method.
- This allows different classes to perform the same task in different ways.

5. What is polymorphism in OOP, and how does it enhance flexibility in code?

Ans: Solved in Chapter

6. How can encapsulation help improve data security in OOP?

Ans: Encapsulation and Data Security

Encapsulation combines data and methods into a single class. It also controls access to data. This helps keep important data safe from unauthorized access.

i. **Protects Data**

- Encapsulation hides important data inside a class.
- It prevents direct access to the data.
- This keeps the data safe.

ii. **Prevents Unauthorized Changes**

- Data cannot be changed directly.
- It can only be changed through specific methods.
- This reduces unwanted changes.

iii. **Improves Security**

- It protects data from unauthorized access.
- It helps maintain the correctness of data.
- This makes the program more secure.

iv. **Controls Data Access**

- The programmer decides how data can be accessed.
- Only approved methods can read or update the data.
- This gives better control over the data.

v. **Makes Programs Easy to Manage**

- Data and related methods stay together in one class.
- This keeps the code organized.
- It also makes the program easier to understand and maintain.

Example

A **BankAccount** class hides the account balance. Users cannot change the balance directly. They can update it only by using methods such as **deposit()** and **withdraw()**. This keeps the account balance secure.

7. How does the combination of inheritance, encapsulation, and polymorphism contribute to better software design?

Ans: Role of Inheritance, Encapsulation, and Polymorphism

Inheritance, encapsulation, and polymorphism are important concepts of Object-Oriented Programming (OOP). Together, they make software more organized, secure, flexible, and easy to maintain.

i. **Inheritance Reuses Code**

- Inheritance allows a child class to use the features of a parent class.
- It reduces code repetition.
- It saves time and effort.
- It makes programs easy to extend.

ii. **Encapsulation Protects Data**

- Encapsulation keeps data and methods together in one class.
- It restricts direct access to important data.
- It protects data from unauthorized changes.
- It improves program security.

iii. Polymorphism Increases Flexibility

- Polymorphism allows the same method to work in different ways.
- Different objects can perform different actions using the same method.
- It makes programs flexible and reusable.

iv. Combined Benefits

- Reduces code duplication.
- Protects important data.
- Makes programs flexible.
- Makes code easy to understand.
- Makes software easier to update and maintain.
- Helps build well-structured programs.

Example

In a School Management System:

- **Inheritance** allows the Student class to reuse features from the Person class.
- **Encapsulation** keeps student data safe inside the class.
- **Polymorphism** allows different classes to use the same method in different ways.

8. What are the key differences between a class and an object in Object-Oriented Programming?

Ans: Solved in chapter

ADDITIONAL SHORT QUESTIONS

Object-Oriented Programming (OOP)

1. What are the benefits of Object-Oriented Programming (OOP)?

Ans: OOP makes programs easy to understand and manage. It reduces code repetition and saves time. It makes programs easier to maintain and update. It also supports scalability and teamwork among programmers.

2. What is modularity in OOP?

Ans: Modularity divides a program into small modules. It makes the program easy to understand. It also makes development easier. It helps programmers manage the program efficiently.

3. How does OOP support scalability and adaptability?

Ans: OOP supports software growth. New features can be added easily. It makes programs flexible. It helps software adapt to new requirements.

4. State the core concepts of OOP?

Ans: The core concepts of OOP are classes and objects. They also include encapsulation, inheritance, and polymorphism. These concepts make programs organized. They help solve real-world problems effectively.

5. What is an object in OOP?

Ans: An object is an instance of a class. It contains actual values for attributes. It represents a real-world entity. Objects can use the methods of their class.

Encapsulation

6. What are the levels of access control in Python?

Ans: Python provides public, protected, and private members. These levels control how data is accessed. They help organize class members. They improve data protection.

7. What are public members in Python?

Ans: Public members can be accessed from anywhere. They are available inside and outside the class. They are public by default in Python. They are easy to use.

8. **What are protected members in Python?**

Ans: Protected members are mainly used within the class and its subclasses. They use a single underscore before the name. They follow a naming convention. They do not strictly prevent access.

9. **What are private members in Python?**

Ans: Private members are intended for use inside the class only. They use double underscores before the name. Python uses name mangling for them. This reduces accidental access from outside the class.

Inheritance

10. **Why is inheritance important in OOP?/ What are the benefits of inheritance?**

Ans: Inheritance makes programs more reusable. It reduces code repetition. It makes programs easier to maintain. It also allows adding new features easily.

11. **What is a parent class in OOP?**

Ans: A parent class is also called a superclass or base class. Its properties and methods are inherited by another class. It provides common features. It is used to create child classes.

12. **What is a child class in OOP?**

Ans: A child class is also called a subclass or derived class. It inherits properties and methods from a parent class. It can also add new features. It extends the functionality of the parent class.

13. **What is single inheritance?**

Ans: Single inheritance occurs when a child class inherits from one parent class. It reuses the features of one class. It is simple and commonly used in Python. It also allows adding new methods.

14. **What is multiple inheritance?**

Ans: Multiple inheritance occurs when a child class inherits from more than one parent class. It combines features of different classes. It increases code reuse. It makes programs more flexible.

15. **What is multilevel inheritance?**

Ans: Multilevel inheritance creates a chain of inheritance. A child class inherits from another child class. It can use features from both parent and grandparent classes. It improves code organization.

16. **What is method overriding in OOP?**

Ans: Method overriding allows a child class to define its own version of a parent class method. It changes the method's behavior when needed. It provides flexibility in programming. It supports customization in child classes.

17. **What is the purpose of the super() function?**

Ans: The super() function is used to call the parent class method. It helps reuse the parent class functionality. It avoids code duplication. It makes the code easier to manage.

18. **How is a child class created in Python?**

Ans: A child class is created by writing the parent class name inside parentheses. It inherits the parent's properties and methods. It can also define its own methods. This makes the class more useful.

Polymorphism

19. **What are the types of polymorphism?**

Ans: Polymorphism has two main types. These are compile-time polymorphism and runtime polymorphism. Both allow methods to behave differently. They improve the flexibility of programs.

20. **How does Python support compile-time polymorphism?**

Ans: Python does not support true method overloading. It uses default arguments and variable-length arguments. A single function can accept different types of input. This provides behavior similar to compile-time polymorphism.

21. **What is method overriding in runtime polymorphism?**

Ans: Method overriding allows a child class to define its own version of a parent class method. The method's behavior changes according to the object. It is used in runtime polymorphism. It improves code flexibility.

22. What are the benefits of polymorphism? / Why is polymorphism important in OOP?

Ans: Polymorphism improves code reuse. It makes programs more flexible. It allows the same method to perform different actions for different objects. It also makes software easier to extend and maintain.

23. What is the pass statement in Python and why it is used?

Ans: The pass statement is a null statement in Python. It does nothing when executed. It is used as a placeholder where a statement is required but no code is written yet.

24. Differentiate between method overloading and method overriding.

Method Overloading	Method overriding
Same method name is used with different parameters / arguments.	A child class defines the same method of its parent class with its own implementation.
Inheritance is not required	Inheritance is required.

DETERMINE OUTPUT SHORT QUESTIONS

Q1. Determine output of following code.

```
class Person:
    def __init__(self):
        print("Person")
class Student(Person):
    def __init__(self):
        super().__init__()
        print("Student")
s = Student()
```

Ans: Output
Person
Student

Q2. Determine output of following code.

```
class Animal:
    def sound(self):
        print("Animal")
class Dog(Animal):
    def sound(self):
        print("Bark")
a = Animal()
d = Dog()
a.sound()
d.sound()
```

Ans: Output
Animal
Bark

Q3. Determine output of following code.

```
class A:
    def show(self):
        print("A")
class B:
    def display(self):
        print("B")
class C(A, B):
    pass
```

```
obj = C()
obj.display()
obj.show()
```

Ans: Output

B

A

Q4. Determine output of following code.

```
class Student:
    def __init__(self):
        self.marks = 95
    def show(self):
        print(self.marks)
s = Student()
s.show()
```

Ans: Output

95

WRITE A CODE SHORT QUESTIONS

1. Write Python code to create a class with a private variable. (Encapsulation)

Ans: class Student:

```
    def __init__(self):
        self.__marks = 90
```

2. Write Python code to inherit the Student class from the Person class. (Single Inheritance)

Ans: class Person:

```
    pass
class Student(Person):
    pass
```

3. Write Python code to create a class that inherits from two parent classes. (Multiple Inheritance)

Ans: class A:

```
    pass
class B:
    pass
class C(A, B):
    pass
```

4. Write Python code to override the display() method in the child class. (Run-Time Polymorphism)

Ans: class Parent:

```
    def display(self):
        print("Parent")
class Child(Parent):
    def display(self):
        print("Child")
```

5. Write Python code to define a method with default arguments. (Compile-Time Polymorphism)

Ans: class Calculator:

```
    def add(self, a=1, b=1):
        return a+b
```

6. Write Python code to call the parent class constructor using the super() function.

Ans: class Person:

```
    def __init__(self):
        print("Person")
class Student(Person):
    def __init__(self):
        super().__init__()
```

7. Write Python code to create an object of the Student class and call its show() method.

Ans:

```
class Student:
    def show(self):
        print("Hello")
s = Student()
s.show()
```

8. Write a single line of code to create a private variable named salary inside a class.

Ans:

```
self.__salary = 50000
```

The double underscore (__) makes the variable private. A private variable cannot be accessed directly from outside the class.

FIND THE ERRORS SQ

1. Find the errors in the following code and then write correct code.

```
class Student
    pass
s1 = Student
```

Ans: **Error 1:** The class declaration is missing a colon (:).

Error 2: Student should be called with parentheses to create an object.

Correct Code:

```
class Student:
    pass
s1 = Student()
```

2. Find the errors in the following code and then write correct code.

```
class Student:
    def _inite_(self):
        print("Student Created")
s1 = Student()
```

Ans: **Error 1:** The constructor name is incorrect. It should be __init__().

Error 2: The constructor must have two underscores before and after init.

Correct Code:

```
class Student:
    def __init__(self):
        print("Student Created")
s1 = Student()
```

3. Find the errors in the following code and then write correct code.

```
class Student:
    def __init__(self):
        self.__marks = 90
s = Student()
print(s.__marks)
```

Ans: **Error 1:** __marks is a private attribute.

Error 2: A private attribute cannot be accessed directly outside the class.

Correct Code:

```
class Student:
    def __init__(self):
        self.__marks = 90
    def show(self):
        print(self.__marks)
s = Student()
s.show()
```

4. Find the errors in the following code and then write correct code.

```
class Animal:
    pass
class Dog Animal:
    pass
```

Ans: **Error 1:** Parent class name must be enclosed in parentheses.
Error 2: The class declaration is missing a colon (:).

Correct Code:

```
class Animal:
    pass
class Dog (Animal):
    pass
```

5. Find the errors in the following code and then write correct code.

```
class A:
    pass
class B:
    pass
class C(A B):
    pass
```

Ans: **Error 1:** Parent class names must be separated by a comma.
Error 2: The class declaration should be class C(A, B).

Correct Code:

```
class A:
    pass
class B:
    pass
class C(A, B):
    pass
```

6. Find the errors in the following code and then write correct code.

```
class Animal:
    def sound(self):
        print("Animal Sound")
class Dog (Animal):
    def sound():
        print("Dog Bark")
```

Ans: **Error 1:** The sound() method in the child class is missing the self parameter.
Error 2: Instance methods must include self as the first parameter.

Correct Code:

```
class Animal:
    def sound(self):
        print("Animal Sound")
class Dog (Animal):
    def sound(self):
        print("Dog Bark")
```

CONCEPTUAL SHORT QUESTIONS

1. If you have a class called **Player** for a video game, write a single line of code to add an attribute for the player's health and set it to 100.
Ans: `self.health = 100`
This line creates a health attribute inside the **Player** class and assigns the initial value 100 to it.
2. A class named **Student** is created. Write a single line of code to create an object of this class.
Ans: `s1 = Student()`
This statement creates an object named **s1** from the **Student** class. The object can access the attributes and methods of the class.
3. You have an object **car1** of class **Car**. Write a single line of code to access the color attribute of the object.
Ans: `car1.color`
This statement accesses the color attribute of the **car1** object. It is used to get the value stored in the object attribute.
4. If a class **BankAccount** contains balance data and methods together, which OOP concept is being used?
Ans: The OOP concept used is encapsulation. It combines data (balance) and methods that operate on the data into a single unit called a class.
5. A programmer wants to hide the password data of a user from direct access. Which OOP concept should be used?
Ans: The programmer should use encapsulation. It protects important data from unwanted access by restricting direct access to class members.
6. A class **Car** inherits from a class **Vehicle**, and another class **ElectricCar** inherits from **Car**. Which type of inheritance is this?
Ans: This is multilevel inheritance. In this type, a class inherits from another derived class, creating a chain of inheritance.
7. A class **Child** inherits features from two classes **Father** and **Mother**. Which type of inheritance is used?
Ans: This is called multiple inheritance. In multiple inheritance, one child class receives properties and methods from more than one parent class.
8. A programmer creates one parent class and one child class. Write the name of the inheritance type used.
Ans: The inheritance type used is single inheritance. It allows a child class to inherit features from one parent class.
9. In languages such as **Java** and **C++**, method overloading is an example of which type of polymorphism?
Ans: Method overloading is an example of compile-time polymorphism. The decision about which method to use is made during compilation.
10. A **Python** program uses default arguments to allow a method to work with different numbers of arguments. What behavior is being demonstrated?
Ans: The concept used is compile-time polymorphism. It allows the same method name to perform different tasks by using different parameters.
11. A parent class method is overridden by a child class method. Which type of polymorphism is used?
Ans: This is run-time polymorphism. The child class provides its own version of the parent class method, and the decision is made during execution.

TEST YOUR SKILLS

Total Marks: 40

Time: 1:30 Hour

8 x 1 = 8

Q.1 MULTIPLE CHOICE QUESTIONS

1. A template used to create many similar objects is called:
A) Method B) Object C) Class D) Variable
2. Which naming style represents a private member in Python?
A) self.name B) self._name C) self__name D) self.name
3. The OOP concept that hides data inside a class is:
A) Inheritance B) Polymorphism C) Encapsulation D) Abstraction
4. Inheritance in OOP means:
A) Reusing and extending functionality from a parent class to a child class.
B) Hiding data in a class.
C) Creating multiple instances of the same class.
D) Managing polymorphic behavior of classes.
5. The keyword used to define a class in Python is:
A) def B) class C) function D) object
6. Polymorphism in OOP is best defined as:
A) Allowing an object to take multiple forms through method overriding.
B) Hiding data inside a class for security.
C) Using multiple inheritance to extend classes.
D) Ensuring that each object has only one method.
7. A class that inherits properties and methods from another class is called:
A) Parent class B) Super class C) Derived class D) Blueprint
8. Runtime polymorphism is achieved through:
A) Encapsulation B) Method overriding C) Multiple inheritance D) Object creation

Q.2 SHORT QUESTIONS:

Note: Attempt any 8 short questions

8 x 2 = 16

- i. What is the difference between a class and an object?
- ii. What are protected members in Python?
- iii. What is the purpose of `__init__()` method?
- iv. How does inheritance enable the extension of functionality in Python?
- v. What is polymorphism in OOP, and how does it make code more flexible?
- vi. How does encapsulation help in keeping data secure and private?
- vii. What is encapsulation in OOP, and why is it important?
- viii. Differentiate between method overloading and method overriding.
- ix. Determine output

```
class Student:
    def __init__(self):
        self.__marks = 95
    def show(self):
        print(self.__marks)
s = Student()
s.show()
```

- x. Write Python code to create a class that inherits from two parent classes.

LONG QUESTIONS:

Note: Attempt any 2 long questions

2 x 8 = 16

3. Write a Python program to demonstrate multiple inheritance using Teacher, Coach, and Student classes.
4. Write a Python program to demonstrate multilevel inheritance using GrandFather, Father, and Son.
5. Write a Python program to demonstrate runtime polymorphism using Animal, Dog, and Cat classes.