## **OOP Notes**

## **What is Encapsulation in OOP?**

Encapsulation is a fundamental concept in object-oriented programming (OOP) that involves bundling the data (attributes) and the methods (functions) that operate on the data into a single unit, known as a class. The main purpose of encapsulation is to protect the data from unauthorized access and modification. This is achieved by restricting access to certain components of an object and only allowing it through well-defined interfaces.

**Key Points of Encapsulation:**

1. **Access Modifiers**: Encapsulation is implemented using access modifiers. Common access modifiers include:
   * **Public**: Members (data and methods) declared as public can be accessed from any other code.
   * **Private**: Members declared as private can only be accessed within the class they are defined.
   * **Protected**: Members declared as protected can be accessed within the class they are defined and by derived classes.
2. **Getter and Setter Methods**: These are public methods used to access private data.
   * **Getter** methods are used to retrieve the value of a private attribute.
   * **Setter** methods are used to set or update the value of a private attribute.
3. **Benefits of Encapsulation**:
   * **Data Hiding**: Internal object details are hidden from the outside, exposing only what is necessary.
   * **Improved Security**: By controlling access to data, encapsulation helps prevent unauthorized access and modifications.
   * **Modularity**: Encapsulation makes it easier to manage complexity by breaking down the system into manageable components.
   * **Ease of Maintenance**: Changes to the encapsulated code can be made with minimal impact on other parts of the program.

**Example in Python:**

class BankAccount:

def \_\_init\_\_(self, balance):

self.\_\_balance = balance # Private attribute

def deposit(self, amount):

if amount > 0:

self.\_\_balance += amount

def withdraw(self, amount):

if 0 < amount <= self.\_\_balance:

self.\_\_balance -= amount

def get\_balance(self):

return self.\_\_balance # Getter method

# Usage

account = BankAccount(1000)

account.deposit(500)

account.withdraw(200)

print(account.get\_balance()) # Output: 1300

In this example, the \_\_balance attribute is private, and access to it is controlled through the deposit, withdraw, and get\_balance methods. This encapsulation ensures that the balance cannot be directly accessed or modified from outside the class, maintaining the integrity of the data.

---------------------------------------------------------------------------------------------------------------------

## **What is Abstraction In OOP?**

Abstraction in object-oriented programming (OOP) is the concept of simplifying complex systems by modeling classes based on essential properties and behaviors, while hiding unnecessary details. It allows programmers to focus on what an object does rather than how it does it. Abstraction is achieved through abstract classes and interfaces, which define a blueprint for other classes.

**Key Points of Abstraction:**

1. **Essential Characteristics**: Abstraction involves identifying the essential characteristics of an object and focusing on what is important for the specific context.
2. **Hiding Details**: It hides the internal implementation details and shows only the functionalities to the user. This helps in reducing complexity and allows for a cleaner and more understandable codebase.
3. **Abstract Classes and Interfaces**:
   * **Abstract Class**: A class that cannot be instantiated and is meant to be subclassed. It can have abstract methods (without implementation) and concrete methods (with implementation).
   * **Interface**: A contract that a class can implement, ensuring that it provides specific behaviors defined by the interface.
4. **Improved Code Maintainability**: By focusing on high-level operations, abstraction helps in maintaining and modifying code without affecting other parts of the system.

**Example in Python:**

Using an abstract class:

from abc import ABC, abstractmethod

class Animal(ABC):

@abstractmethod

def make\_sound(self):

pass

def sleep(self):

print("Sleeping...")

class Dog(Animal):

def make\_sound(self):

print("Bark")

class Cat(Animal):

def make\_sound(self):

print("Meow")

# Usage

dog = Dog()

dog.make\_sound() # Output: Bark

dog.sleep() # Output: Sleeping...

cat = Cat()

cat.make\_sound() # Output: Meow

cat.sleep() # Output: Sleeping...

In this example:

* The Animal class is an abstract class with an abstract method make\_sound.
* The Dog and Cat classes inherit from Animal and provide implementations for the make\_sound method.
* The sleep method is a concrete method in the Animal class that can be used by any subclass.

By using abstraction, the Animal class provides a blueprint for other classes, ensuring that any subclass of Animal must implement the make\_sound method. This allows the user to work with the Animal type without needing to know the specific details of each subclass.

---------------------------------------------------------------------------------------------------------------------

## **What is Polymorphism in OOP?**

Polymorphism is a core concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different underlying forms (data types). There are two main types of polymorphism: compile-time (static) and runtime (dynamic).

**Key Points of Polymorphism:**

1. **Compile-time Polymorphism (Static Polymorphism)**: This is achieved through method overloading or operator overloading.
   * **Method Overloading**: Multiple methods with the same name but different parameters.
   * **Operator Overloading**: Defining custom behavior for operators (e.g., +, -) for user-defined data types.
2. **Runtime Polymorphism (Dynamic Polymorphism)**: This is achieved through method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass.
   * **Method Overriding**: A subclass implements a method that is already defined in its superclass with the same method signature.
3. **Benefits of Polymorphism**:
   * **Code Reusability**: Common interfaces can be used for different implementations, promoting code reuse.
   * **Flexibility and Maintainability**: New classes can be introduced without modifying existing code, making the system more flexible and easier to maintain.
   * **Dynamic Behavior**: Objects can be used interchangeably if they share a common interface, allowing for dynamic method binding at runtime.

**Example in Python:**

Using runtime polymorphism with method overriding:

class Animal:

def make\_sound(self):

raise NotImplementedError("Subclass must implement abstract method")

class Dog(Animal):

def make\_sound(self):

return "Bark"

class Cat(Animal):

def make\_sound(self):

return "Meow"

class Cow(Animal):

def make\_sound(self):

return "Moo"

# Usage

def animal\_sound(animal):

print(animal.make\_sound())

# Creating objects of different classes

dog = Dog()

cat = Cat()

cow = Cow()

# Passing objects to the function

animal\_sound(dog) # Output: Bark

animal\_sound(cat) # Output: Meow

animal\_sound(cow) # Output: Moo

In this example:

* The Animal class defines an abstract method make\_sound.
* The Dog, Cat, and Cow classes each override the make\_sound method to provide their specific implementations.
* The animal\_sound function demonstrates polymorphism by accepting an Animal type and calling the make\_sound method, which will execute the appropriate implementation based on the object passed.

By using polymorphism, the animal\_sound function can operate on any object that is a subclass of Animal, without needing to know the specific type of the animal. This makes the code more flexible and easier to extend with new animal types.

---------------------------------------------------------------------------------------------------------------------

## **What is Inheritance in OOP?**

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (referred to as a subclass or derived class) to inherit properties and behaviors (methods) from another class (referred to as a superclass or base class). Inheritance promotes code reuse and establishes a natural hierarchy between classes.

**Key Points of Inheritance:**

1. **Code Reusability**: Inheritance allows a subclass to reuse code from its superclass, reducing redundancy and improving maintainability.
2. **Hierarchical Classification**: It establishes a natural hierarchy among classes, promoting a clearer and more logical structure.
3. **Method Overriding**: Subclasses can provide specific implementations of methods that are already defined in their superclass. This allows for dynamic behavior and polymorphism.
4. **Extensibility**: New functionality can be added to existing classes by creating subclasses, making the system more extensible.
5. **Types of Inheritance**:
   * **Single Inheritance**: A subclass inherits from one superclass.
   * **Multiple Inheritance**: A subclass inherits from more than one superclass.
   * **Multilevel Inheritance**: A subclass inherits from a superclass, which in turn inherits from another superclass.
   * **Hierarchical Inheritance**: Multiple subclasses inherit from a single superclass.
   * **Hybrid Inheritance**: A combination of two or more types of inheritance.

**Example in Python:**

Using single inheritance:

­class Animal:

def \_\_init\_\_(self, name):

self.name = name

def make\_sound(self):

raise NotImplementedError("Subclass must implement abstract method")

def move(self):

print(f"{self.name} is moving")

class Dog(Animal):

def make\_sound(self):

return "Bark"

def fetch(self):

print(f"{self.name} is fetching")

# Usage

dog = Dog("Buddy")

print(dog.make\_sound()) # Output: Bark

dog.move() # Output: Buddy is moving

dog.fetch() # Output: Buddy is fetching

In this example:

* The Animal class is the superclass, defining common attributes and methods like name, make\_sound, and move.
* The Dog class is a subclass that inherits from Animal and provides a specific implementation of the make\_sound method. It also adds a new method fetch specific to the Dog class.
* The Dog object dog can access methods from both the Dog class and the Animal class.

Inheritance enables the Dog class to reuse and extend the functionality of the Animal class, promoting code reuse and a clear hierarchical structure.

---------------------------------------------------------------------------------------------------------------------

## **What is the virtual function Give Example?**

A virtual function in object-oriented programming is a function or method defined in a base class that can be overridden in a derived class. The key feature of virtual functions is that they enable polymorphism, allowing a program to decide at runtime which function to invoke based on the type of the object, rather than the type of the reference or pointer.

Here's an example in C++:

#include <iostream>

using namespace std;

class Base {

public:

virtual void show() { // Virtual function

cout << "Base class" << endl;

}

};

class Derived : public Base {

public:

void show() override { // Overriding the virtual function

cout << "Derived class" << endl;

}

};

int main() {

Base\* b;

Derived d;

b = &d;

// This will call the show() function in Derived class

b->show();

return 0;

}

In this example:

1. **Base class** has a virtual function show().
2. **Derived class** overrides the show() function.
3. In the main() function, a pointer of type Base points to an object of type Derived.
4. When b->show() is called, the show() function in the Derived class is invoked, demonstrating runtime polymorphism.

If the show() function in the Base class was not declared as virtual, the show() function in the Base class would be called, not the one in the Derived class. This is the fundamental concept behind virtual functions and polymorphism in object-oriented programming.

---------------------------------------------------------------------------------------------------------------------

## **What is pure virtual function?**

A pure virtual function is a virtual function that does not have an implementation in the base class and must be overridden in any derived class. Declaring a function as pure virtual is a way of enforcing that derived classes provide an implementation for this function, making the base class abstract. This means you cannot create an instance of the base class; it can only be used as a base for other classes.

A pure virtual function is declared by assigning 0 to the virtual function in the base class. Here's an example in C++:

#include <iostream>

using namespace std;

class Base {

public:

virtual void show() = 0; // Pure virtual function

};

class Derived : public Base {

public:

void show() override {

cout << "Derived class" << endl;

}

};

int main() {

Base\* b;

Derived d;

b = &d;

// This will call the show() function in Derived class

b->show();

return 0;

}

In this example:

1. **Base class** has a pure virtual function show().
2. **Derived class** overrides the show() function with its own implementation.
3. In the main() function, a pointer of type Base points to an object of type Derived.
4. When b->show() is called, the show() function in the Derived class is invoked.

Attempting to instantiate an object of the Base class directly would result in a compilation error, as Base is now an abstract class due to the pure virtual function. This ensures that any concrete class derived from Base provides its own implementation of the show() function.