MD. ARSHAD </p>

ALISHARMEEN02@GMAIL.COM </p>

Q1. Explain Class and Object with respect to Object-Oriented Programming. Give a suitable example.

In object-oriented programming (OOP), a class and an object are fundamental concepts that facilitate the organization and structuring of code. They are used to model real-world entities, allowing you to create modular and reusable code.

Class: A class is a blueprint or template for creating objects. It defines the structure and behavior of objects of that type. Think of a class as a blueprint for a specific category of objects, describing their attributes (data) and methods (functions or actions) that they can perform. In essence, a class defines the characteristics and behavior that its objects will possess.

Object: An object is an instance of a class. It's a concrete representation created based on the class blueprint. Objects have state (attributes or properties) and behavior (methods) defined by the class. They can interact with each other and perform operations based on the methods defined in their class.

Here's an example to help illustrate the concepts of class and object:

Example: Class and Object for a Car

python Copy code

Define a Car class

In this example, Car is a class that serves as a blueprint. It has attributes like make, model, year, and a couple of methods like start, stop, and honk. The objects car1 and car2 are instances of the Car class. They have their own distinct attributes and can perform actions based on the methods defined in the class.

The start, stop, and honk methods are actions that can be performed on individual car objects. By creating and using objects, you can model and simulate real-world entities in your code, making it more organized and maintainable.

In [2]:
```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.is_running = False

    def start(self):
        self.is_running = True
        print(f"{self.year} {self.make} {self.model} started.")

    def stop(self):
        self.is_running = False
        print(f"{self.year} {self.make} {self.model} stopped.")

    def honk(self):
        print("Honk! Honk!")

# Create objects (instances) of the Car class
car1 = Car("Toyota", "Corolla", 2023)
car2 = Car("Ford", "Mustang", 2023)

# Use the objects' methods
car1.start()
car2.start()
car1.honk()
car2.stop()
```

```
2023 Toyota Corolla started.
2023 Ford Mustang started.
Honk! Honk!
2023 Ford Mustang stopped.
```

Q2. Name the four pillars of OOPs.

Certainly, the four pillars of Object-Oriented Programming (OOP) are:

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

Q3. Explain why the **init**() function is used. Give a suitable example.

The **init**() function, also known as the constructor, is a special method in object-oriented programming languages like Python. It is used to initialize the attributes (data members) of an object when an instance of a class is created. The **init**() method is automatically called when you create an object of a class, and it allows you to set up the initial state of the object.

Key points about the **init**() function:

Initialization: The primary purpose of the **init**() method is to initialize the attributes of an object with appropriate values. It helps ensure that the object is in a valid and consistent state as soon as it is created.

Automatic Invocation: When you create an object using the class constructor (e.g., my_object = MyClass()), the **init**() method is automatically called. You can provide arguments to the constructor, and these arguments are passed to the **init**() method to initialize the object's attributes.

Self Parameter: The self parameter is a reference to the instance of the class. It is automatically passed as the first parameter to every method, including **init**(). This parameter allows you to access and modify the object's attributes within the method.

Attribute Assignment: Inside the **init**() method, you can use the self parameter to assign values to the object's attributes. These attributes can be used to store data that represents the object's state.

Here's an example to illustrate the use of the **init**() method:

In [3]:
```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print(f"{self.name} says Woof!")

# Create instances of the Dog class
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)

# Access attributes and call methods
print(f"{dog1.name} is {dog1.age} years old.")
dog1.bark()

print(f"{dog2.name} is {dog2.age} years old.")
dog2.bark()
```

```
Buddy is 3 years old.
Buddy says Woof!
Max is 5 years old.
Max says Woof!
```

In this example, the Dog class has an **init**() method that initializes the name and age attributes of each Dog object. When you create instances of the Dog class, you pass values for name and age to the constructor, which then initializes the attributes accordingly. The bark() method is also defined to allow the dogs to bark.

Using the **init**() method ensures that every Dog object is properly initialized with its name and age when it is created. This promotes consistent and predictable behavior in your code.

Q4. Why self is used in OOPs?

In object-oriented programming (OOP), the self keyword is used to refer to the current instance of a class. It is a common convention in languages like Python, and it plays a crucial role in allowing methods to access and manipulate the attributes and behaviors of the object they belong to. The self parameter is automatically passed to instance methods when they are called, and it serves several important purposes: Instance Reference: The self parameter provides a reference to the current instance of the class. This allows methods to interact with the specific attributes and methods of that instance. Without self, methods would not have a way to distinguish which instance they are operating on. Attribute Access: By using self, methods can access and modify the attributes (data members) of the object. For example, you can assign values to attributes, read their values, and perform calculations based on them. Method Invocation: self allows methods to call other methods within the same class. This is especially useful when methods need to collaborate or share functionality. Instance Creation: When an instance of a class is created, the self parameter ensures that the newly created instance is correctly passed to the __init__() method, allowing the object's attributes to be initialized. Here's a simple example to illustrate the use of self in Python:

In [5]:
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        print(f"Hi, my name is {self.name} and I am {self.age} years old.")

# Creating an instance of the Person class
person1 = Person("MOHAMMAD ARSHAD", 26)

# Calling the introduce method
person1.introduce()
```

```
Hi, my name is MOHAMMAD ARSHAD and I am 26 years old.
```

In the example above, self is used in the **init**() method to set the attributes name and age for each Person object. It is also used in the introduce() method to access the name and age attributes when printing the introduction.

Using self ensures that methods can work with the specific instance they are called on, allowing for encapsulation and proper interaction between attributes and methods of an object.

Q5. What is inheritance? Give an example for each type of inheritance.

Inheritance is a key concept in object-oriented programming (OOP) where one class (the subclass or derived class) inherits attributes and behaviors from another class (the superclass or base class). Inheritance promotes code reuse, allows for the creation of more specialized classes, and establishes a hierarchy of classes.

There are different types of inheritance, each serving specific purposes:

Single Inheritance: In single inheritance, a subclass inherits from a single superclass. This forms a linear inheritance chain. A subclass can extend the attributes and methods of its superclass, adding its own unique characteristics. Example of single inheritance:

In [6]:
```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

dog = Dog()
cat = Cat()

print(dog.speak())  # Output: Woof!
print(cat.speak())  # Output: Meow!
```

```
Woof!
Meow!
```

Multiple Inheritance: In multiple inheritance, a subclass inherits from more than one superclass. This allows a subclass to inherit attributes and methods from multiple classes, leading to more complex class relationships. Example of multiple inheritance:

In [7]:
```
class Bird:
    def fly(self):
        return "Flying"

class Mammal:
    def run(self):
        return "Running"

class Bat(Bird, Mammal):
    pass

bat = Bat()

print(bat.fly())  # Output: Flying
print(bat.run())  # Output: Running
```

```
Flying
Running
```

Multilevel Inheritance: In multilevel inheritance, a chain of inheritance is formed, where a subclass inherits from a superclass, and another subclass inherits from the first subclass. This creates a hierarchical structure. Example of multilevel inheritance:

In [8]:
```
class Animal:
    def speak(self):
        pass

class Mammal(Animal):
    pass

class Dog(Mammal):
    def speak(self):
        return "Woof!"

dog = Dog()

print(dog.speak())  # Output: Woof!
```

```
Woof!
```

Hierarchical Inheritance: In hierarchical inheritance, multiple subclasses inherit from a single superclass. Each subclass can have its own additional attributes and methods while sharing the common ones from the superclass. Example of hierarchical inheritance:

In [12]:
```
class Shape:
    def area(self):
        pass

class Circle(Shape):
    def area(self, radius):
        return 3.14 * radius * radius

class Square(Shape):
    def area(self, side):
        return side * side

circle = Circle()
square = Square()

print(circle.area(5))  # Output: 78.5
print(square.area(4))  # Output: 16
#These examples demonstrate different types of inheritance,
#each showcasing how subclasses inherit and extend attributes and
#methods from their respective superclasses.
#Inheritance is a powerful mechanism that enables code reuse and
#supports the creation of organized and modular class hierarchies.
```

```
78.5
16
```

Thank You ,That's All </p>