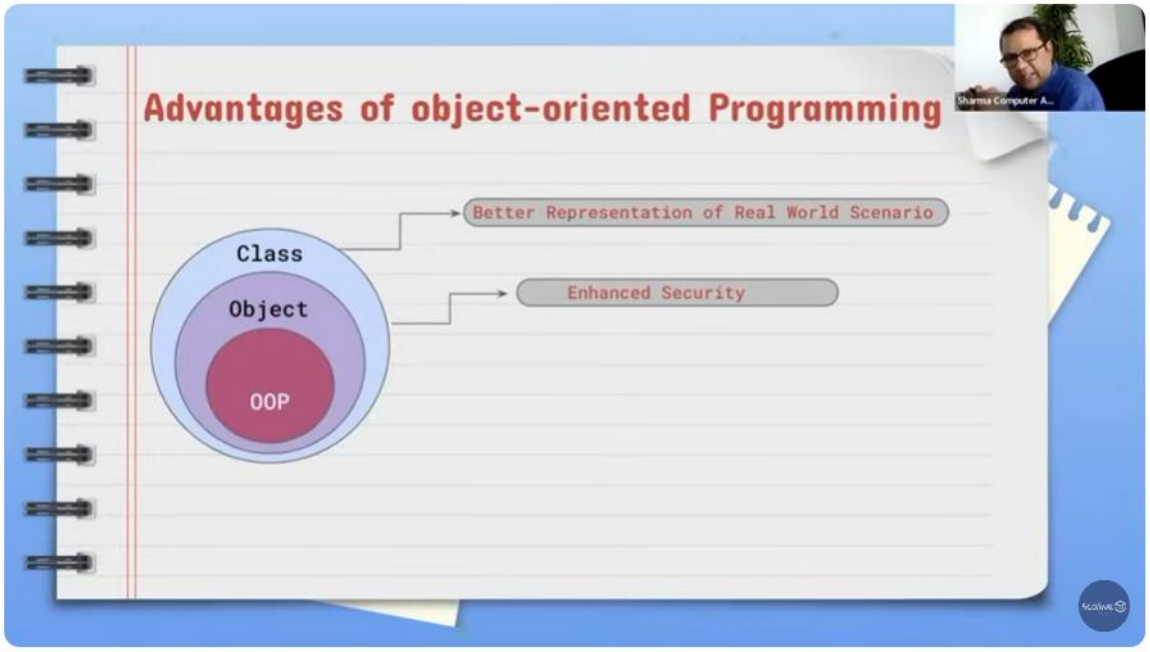
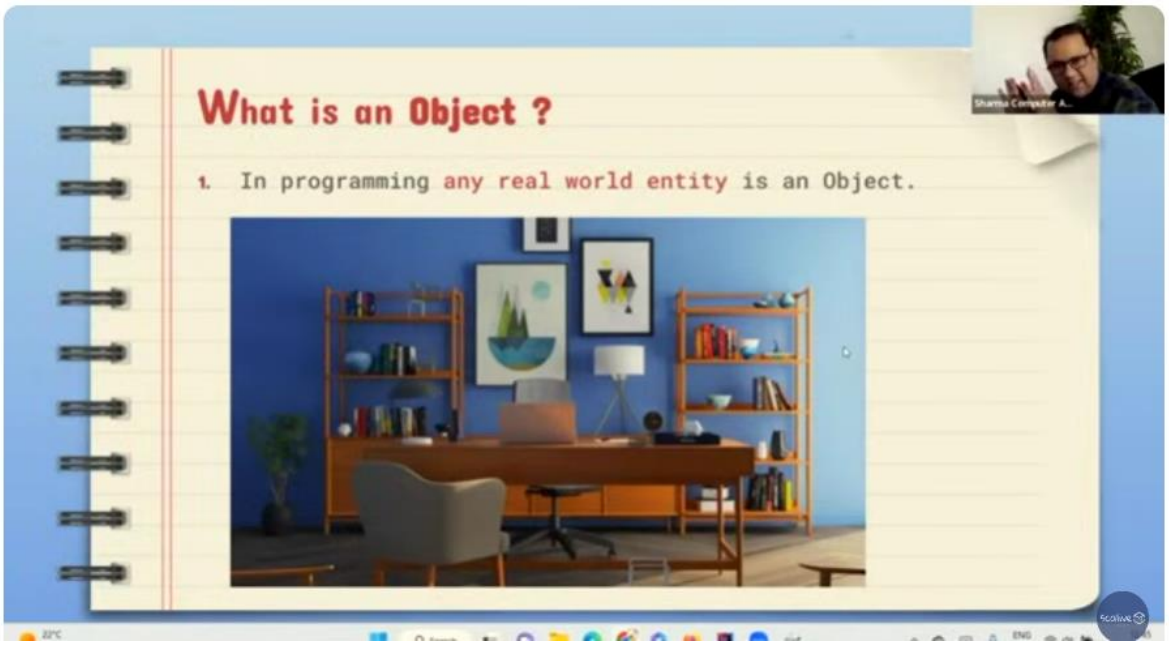
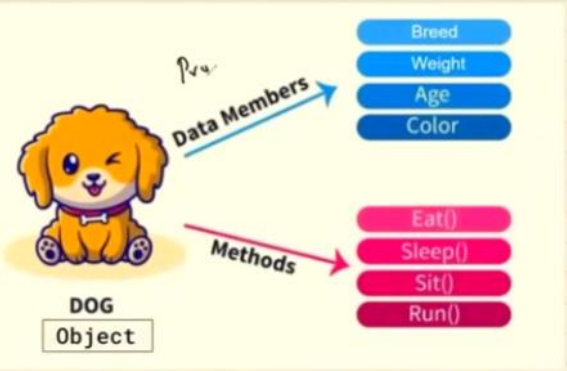

Advantages of object-oriented Programming
Class
Object
OOP
Better Representation of Real World Scenario
Enhanced Security


What is an Object ?
1. In programming any real world entity is an Object.

Components of an Object ?

1. Every Object has two main components :- Property and Behaviours



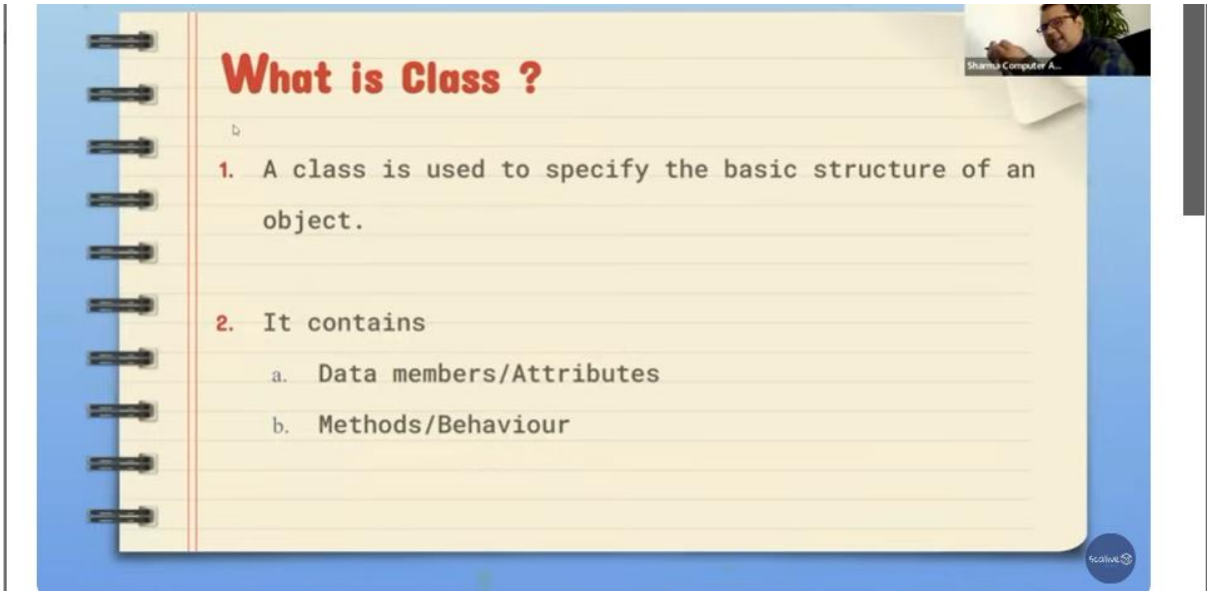
Question -

Are you an Object ?

Yes,we humans are objects because:

We have attributes as name, height, age etc.

We also can show behaviors like walking, talking, running, eating etc.



Keys of OOPS:

1.class

2.object: data member is attribute and methods are actions or functions

class is a object data type, thus we always require to make our own data types and then we use it for objects

class is always logical(humans) and object is always physical(particular human like Ramesh , Suresh)

3.encapsulation

4.Abstraction

5.polymorphism

6.Abstraction

Object-Oriented Programming (OOP) offers several advantages over Procedural Programming (POP) in terms of security and code organization. Here are some key ways in which OOP enhances security compared to POP:

1. **Encapsulation**

- **OOP**: Encapsulation is one of the core principles of OOP. It involves bundling data and methods that operate on that data into a single unit called a class. By using access modifiers (e.g., private, protected), OOP allows you to hide the internal state of an object and only expose what is necessary through public methods.

- **Example**: You can define a class with private attributes and public methods to access or modify those attributes. This prevents direct access to the internal state, reducing the risk of unintended modifications.

```python
class Account:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):
        return self.__balance

acc = Account(100)
acc.deposit(50)
print(acc.get_balance())  # Output: 150
```

- **POP**: In procedural programming, data and functions are separate. There is no built-in mechanism to restrict access to data, making it easier for unintended modifications or misuse.

2. **Abstraction**

- **OOP**: Abstraction allows you to define complex systems in a simplified manner by exposing only relevant details and hiding the internal complexities. This reduces the risk of misuse by focusing on what an object does rather than how it does it.

- **Example**: An abstract class can define an interface, and derived classes can implement the actual functionality.

```python
from abc import ABC, abstractmethod
```

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

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

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

circle = Circle(5)
print(circle.area())  # Output: 78.5
```

- **POP**: Procedural programming typically involves working directly with functions and data structures, making it harder to hide implementation details and potentially exposing internal workings to unauthorized access.

3. **Inheritance**

- **OOP**: Inheritance allows classes to inherit properties and methods from other classes. This promotes code reuse and helps in building a hierarchical structure where security constraints can be managed at different levels.

 - **Example**: A base class can define common functionality, while derived classes can override or extend this functionality as needed.

```python
class Employee:
    def __init__(self, name):
        self.__name = name

    def get_name(self):
```

```
        return self.__name

    class Manager(Employee):
        def __init__(self, name, department):
            super().__init__(name)
            self.__department = department

        def get_department(self):
            return self.__department

    mgr = Manager('Alice', 'HR')
    print(mgr.get_name())  # Output: Alice
    print(mgr.get_department())  # Output: HR
    ```
```

- **POP**: Procedural programming does not support inheritance directly, making it harder to build reusable and secure code structures.

### 4. **Polymorphism**

- **OOP**: Polymorphism allows objects to be treated as instances of their parent class rather than their actual class. This means that you can use a unified interface to interact with different objects, which can enhance security by limiting the exposure of specific implementations.

  - **Example**: Different classes can implement a common interface or method in their own way, but the user interacts with them through the common interface.

    ```python
    class Bird:
        def make_sound(self):
            return "Chirp"

    class Dog:
        def make_sound(self):
```
```

```
        return "Woof"

def print_sound(animal):
    print(animal.make_sound())

print_sound(Bird())  # Output: Chirp
print_sound(Dog())   # Output: Woof
```

- **POP**: In procedural programming, functions operate on data and do not provide a mechanism for polymorphism. This makes it less flexible in handling different types of data uniformly.

Summary

OOP provides enhanced security and code organization through:

- **Encapsulation**: Hides internal state and only exposes necessary functionalities.
- **Abstraction**: Simplifies complex systems by focusing on high-level functionalities.
- **Inheritance**: Promotes code reuse and manages security constraints through class hierarchies.
- **Polymorphism**: Allows a unified interface for interacting with different types of objects, reducing exposure to implementation details.

In contrast, procedural programming lacks these features, making it more challenging to manage and secure complex systems.

NOTES:

object:datamember is attribute and methods are actions or functions

class is a object data type , thus we always require to make our own data types and then we use it for objects

class is always logical(humans) and object is always physical(particalar human like ramesh,suresh)

You've touched on several important concepts in Object-Oriented Programming (OOP). Let me clarify and expand on these points:

Attributes and Methods

- **Attributes (Data Members)**: In OOP, attributes (also known as data members or properties) are variables that belong to an object. They represent the state or characteristics of the object. For instance, if you have a `Person` class, the attributes might include `name`, `age`, and `address`.

Example:

```python
class Person:
    def __init__(self, name, age):
        self.name = name  # Attribute
        self.age = age    # Attribute
```

- **Methods (Functions)**: Methods are functions that belong to an object. They define the actions or behaviors that an object can perform. For example, a `Person` class might have methods like `greet()` or `celebrate_birthday()`.

Example:

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

    def greet(self):
        return f"Hello, my name is {self.name}."

    def celebrate_birthday(self):
        self.age += 1
        return f"Happy Birthday {self.name}! You are now {self.age} years old."
```

Class and Object

- **Class**: A class is a blueprint for creating objects. It defines a type of object and the common attributes and methods that objects of this type will have. The class itself is a logical construct, representing a general concept or category.

  Example:

  ```python
  class Car:
      def __init__(self, make, model):
          self.make = make
          self.model = model

      def start_engine(self):
          return f"The {self.make} {self.model}'s engine is now running."
  ```

- **Object**: An object is an instance of a class. It represents a specific realization of the class. While the class is a logical concept, objects are concrete instances with actual data. For example, `my_car` could be an object of the `Car` class, with specific values for `make` and `model`.

  Example:

  ```python
  # Creating an object of the Car class
  my_car = Car(make="Toyota", model="Corolla")

  # Using methods of the Car class on the object
  print(my_car.start_engine())  # Output: The Toyota Corolla's engine is now running.
  ```

Summary

- **Attributes**: Represent the state or characteristics of an object.

- **Methods**: Define the actions or behaviors of an object.

- **Class**: A blueprint for creating objects, defining common attributes and methods.

- **Object**: A specific instance of a class, representing a particular realization of the class.

Your understanding that a class is a logical construct and objects are concrete instances is accurate. Classes provide a way to model real-world entities (like "Person" or "Car"), and objects are specific instances of these models (like "Ramesh" or "Suresh" for the `Person` class). This distinction allows OOP to encapsulate data and functionality in a way that mirrors real-world systems and relationships.

Methods in OOP

In Object-Oriented Programming (OOP), methods are functions that are defined within a class and operate on instances of that class (i.e., objects). Methods can modify the object's state and perform actions using the data within the object.

- **`self` Parameter**: The `self` parameter in methods refers to the instance of the class on which the method is called. It allows access to the object's attributes and other methods from within the class.

Examples of Methods:

1. **Basic Method Example:**

```python
class Dog:
    def __init__(self, name):
        self.name = name

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

# Create an object of the Dog class
my_dog = Dog(name="Buddy")
print(my_dog.bark())  # Output: Buddy says Woof!
```

```

2. **Method Modifying Object State:**

```python
class Counter:
    def __init__(self):
        self.count = 0

    def increment(self):
        self.count += 1

    def get_count(self):
        return self.count

# Create an object of the Counter class
counter = Counter()
counter.increment()
print(counter.get_count())  # Output: 1
```

3. **Method Using Multiple Attributes:**

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height
```
```

```
    def perimeter(self):
        return 2 * (self.width + self.height)

# Create an object of the Rectangle class
rect = Rectangle(5, 3)
print(rect.area())      # Output: 15
print(rect.perimeter())  # Output: 16
```

Constructor in OOP

A constructor is a special method that is automatically called when an object is created. It is used to initialize the object's attributes.

- **Constructor Name**: In Python, the constructor method is named `__init__`. It is not called directly; instead, it is called automatically when a new object of the class is created.

Examples of Constructors:

1. **Basic Constructor Example:**

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

# Create an object of the Person class
person1 = Person(name="Alice", age=30)
print(person1.name)  # Output: Alice
print(person1.age)   # Output: 30
```

```

2. **Constructor with Default Values:**

```python
class Car:
    def __init__(self, make, model, year=2020):
        self.make = make
        self.model = model
        self.year = year

# Create an object of the Car class with default year
car1 = Car(make="Toyota", model="Corolla")
print(car1.year)  # Output: 2020

# Create an object of the Car class with specified year
car2 = Car(make="Honda", model="Civic", year=2022)
print(car2.year)  # Output: 2022
```

3. **Constructor with Complex Initialization:**

```python
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def description(self):
        return f"'{self.title}' by {self.author}, {self.pages} pages."
```
```

```
# Create an object of the Book class
book1 = Book(title="1984", author="George Orwell", pages=328)
print(book1.description())  # Output: '1984' by George Orwell, 328 pages.
```

Creating Objects in Python

In Python, you can create objects in several ways, typically by instantiating classes. Here are two examples of each method:

1. **Creating an Object Directly:**

```python
class Dog:
    def __init__(self, name):
        self.name = name

# Create an object of the Dog class
my_dog = Dog(name="Max")
```

2. **Creating an Object Using a Factory Method:**

```python
class Cat:
    @staticmethod
    def create_black_cat(name):
        return Cat(name, color="black")

    def __init__(self, name, color="gray"):
```

```
        self.name = name
        self.color = color

    # Create an object using the factory method
    black_cat = Cat.create_black_cat(name="Whiskers")
    ```
```

### Why `self` is Used

The `self` parameter in methods allows access to the instance of the class. It is used to refer to instance variables and methods within the class. Without `self`, you cannot access or modify the instance's attributes or call other methods on that instance.

**Examples:**

1. **Accessing Instance Variables:**

```
    ```python
    class Car:
        def __init__(self, make, model):
            self.make = make
            self.model = model

        def display_info(self):
            return f"Car: {self.make} {self.model}"

    car = Car("Toyota", "Corolla")
    print(car.display_info())  # Output: Car: Toyota Corolla
    ```
```

2. **Modifying Instance Variables:**
```

```python
class BankAccount:
    def __init__(self, balance):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def get_balance(self):
        return self.balance

account = BankAccount(100)
account.deposit(50)
print(account.get_balance())  # Output: 150
```

Passing Arguments in Python

Arguments can be passed to functions in various ways:

1. **Positional Arguments:**

```python
def greet(name, age):
    return f"Hello {name}, you are {age} years old."

print(greet("Alice", 30))  # Output: Hello Alice, you are 30 years old.
```

2. **Keyword Arguments:**

```python
def greet(name, age):
    return f"Hello {name}, you are {age} years old."

print(greet(age=30, name="Bob"))  # Output: Hello Bob, you are 30 years old.
```

3. **Default Arguments:**

```python
def greet(name, age=25):
    return f"Hello {name}, you are {age} years old."

print(greet("Carol"))  # Output: Hello Carol, you are 25 years old.
```

4. **Variable-Length Arguments:**

```python
def greet(*names):
    return f"Hello {', '.join(names)}!"

print(greet("Alice", "Bob", "Charlie"))  # Output: Hello Alice, Bob, Charlie!
```

Constructor Calling

Yes, a constructor is called automatically when an object is created. The `__init__` method is invoked when you instantiate a class, which initializes the object's attributes.

Example:

```python
class Laptop:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def display_info(self):
        return f"Laptop: {self.brand} {self.model}"

# Create an object of the Laptop class
laptop = Laptop("Dell", "XPS 13")
print(laptop.display_info())  # Output: Laptop: Dell XPS 13
```

Inheritance in Python

Inheritance is a feature of OOP that allows a class (called the subclass or derived class) to inherit attributes and methods from another class (called the superclass or base class). It promotes code reuse and establishes a hierarchical relationship between classes.

Types of Inheritance:

1. **Single Inheritance:**
   - **Example**: A `Dog` class inherits from an `Animal` class.

     ```python
     class Animal:
         def speak(self):
             return "Animal speaks"

     class Dog(Animal):
     ```

```
    def bark(self):
        return "Woof!"

dog = Dog()
print(dog.speak())  # Output: Animal speaks
print(dog.bark())   # Output: Woof!
```

2. **Multiple Inheritance:**
   - **Example**: A `Smartphone` class inherits from both `Phone` and `Camera` classes.

```python
class Phone:
    def call(self):
        return "Calling"

class Camera:
    def capture(self):
        return "Capturing photo"

class Smartphone(Phone, Camera):
    def browse(self):
        return "Browsing the internet"

smartphone = Smartphone()
print(smartphone.call())     # Output: Calling
print(smartphone.capture())  # Output: Capturing photo
print(smartphone.browse())   # Output: Browsing the internet
```

3. **Multilevel Inheritance:**

- **Example**: A `Grandchild` class inherits from a `Child` class, which in turn inherits from a `Parent` class.

```python
class Parent:
    def speak(self):
        return "Parent speaks"

class Child(Parent):
    def play(self):
        return "Child plays"

class Grandchild(Child):
    def study(self):
        return "Grandchild

studies"

grandchild = Grandchild()
print(grandchild.speak())  # Output: Parent speaks
print(grandchild.play())   # Output: Child plays
print(grandchild.study())  # Output: Grandchild studies
```

4. **Hierarchical Inheritance:**
- **Example**: Both `Dog` and `Cat` classes inherit from an `Animal` class.

```python
class Animal:
    def eat(self):
        return "Animal eats"
```

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

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

dog = Dog()
cat = Cat()
print(dog.eat())  # Output: Animal eats
print(dog.bark()) # Output: Woof!
print(cat.eat())  # Output: Animal eats
print(cat.meow()) # Output: Meow!
```

Inheritance allows classes to share and extend functionalities, which promotes a more organized and reusable codebase.