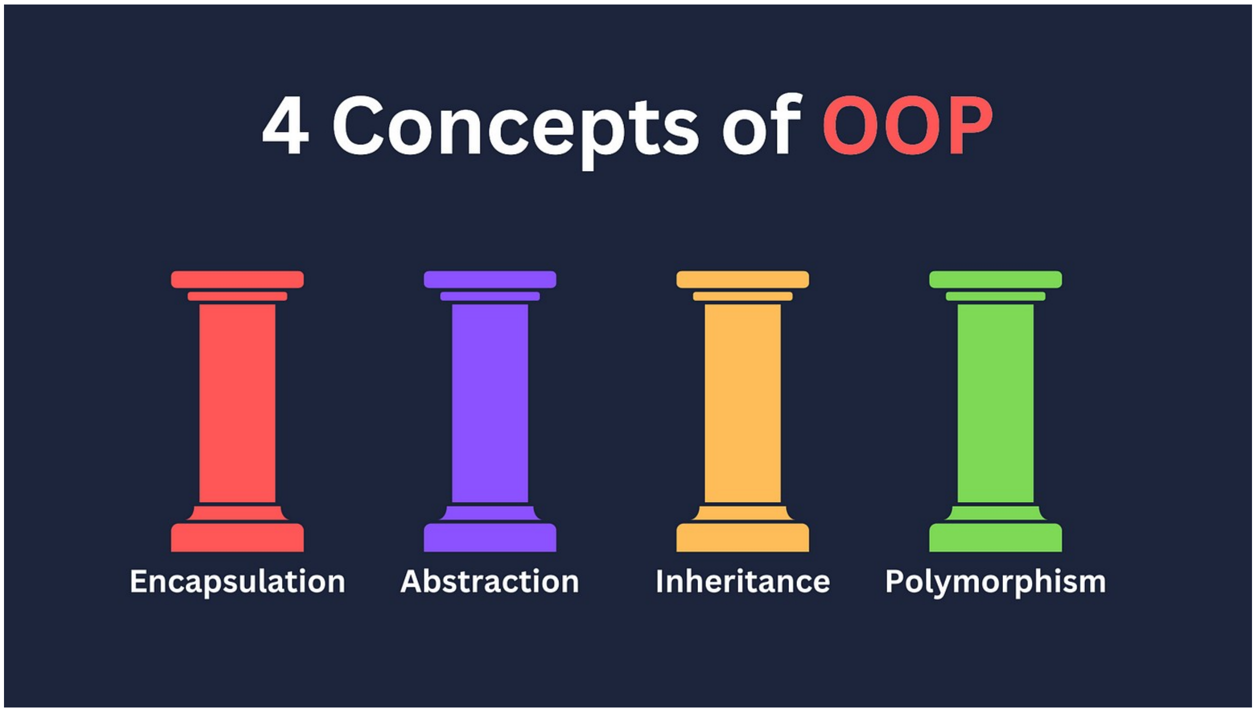
Object Oriented Programming (OOPs)👩💻

OOPs is a programming paradigm that works on four principles;

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

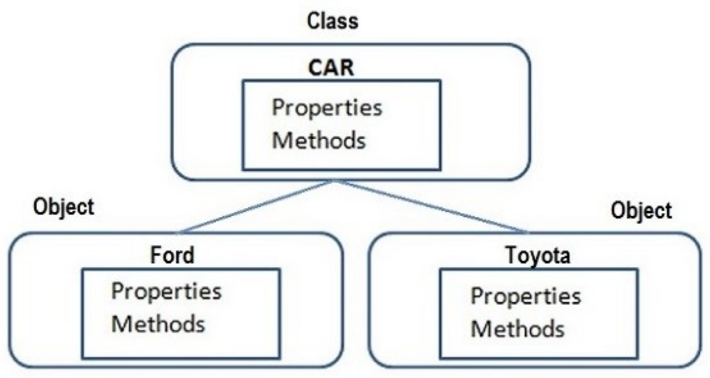
The basic concept of OOPs is to create an object, reuse them throughout the program & manipulate objects to get desire result



Classes

- Classes are group of same entities.
- Class is like a blueprint of the object.
- Classes contains fields & method in it.
- For eg. If there is person class which contains the name & age as properties and the method performed with these person are walking, talking, eating, etc.

```
#Create a class using the class keyword.
class Details:
    name = "Zahid"
    address = "Mumbai"
    age = 23
```



Objects

- It is an instance of class, there can be multiple instance of a class in a program.
- An object is nothing but a self contained component which consists of method & properties to make a particular type of data useful

```
class Details:
    name = "Zahid"
    address = "Mumbai"
    age = 23

obj1 = Details()
print(obj1.name)
print(obj1.address)
print(obj1.age)

Zahid
Mumbai
23
```

Constructor

- It is a special member function of class.
- Basically constructor is automactically called when an object is created.
- It don't have any return types.
- In python **init** is consider as constructor.
- The name of the constructor should be same as the class name.
- There are 3 types of constructors:
  - Default Constructor:- With no parameter
  - Parametric Constructor:- With Parameter. It creates an object and pass parameter simultaneously
  - Copy Constructor:- It creates a new object as a copy of an existing object

init method

The init method is used to initialize the object's state and contains statements that are executed at the time of object creation.

```
class Person:
    def __init__(self, name, age, address): # inbuilt python 
Constructor
        self.name = name
        self.address = address
        self.age = age
p1 = Person("Zahid", 23, "Mumbai") # Creating object p1 of class 
Person

print(p1.name)
print(p1.age)
print(p1.address)
```

```
Zahid
23
Mumbai
```

self method / self Parameter

- The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.
- It must be provided as the extra parameter inside the method definition.
- It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class

```
class Details:
    name = "Zahid"
    address = "Mumbai"
    age = 23

    def desc(self):
        print("My name is", self.name, 
              ", I'm from",self.address, 
              "and I'm", self.age, "years old.")

obj1 = Details()
obj1.desc()
```

```
My name is Zahid , I'm from Mumbai and I'm 23 years old.
```

The str() Function

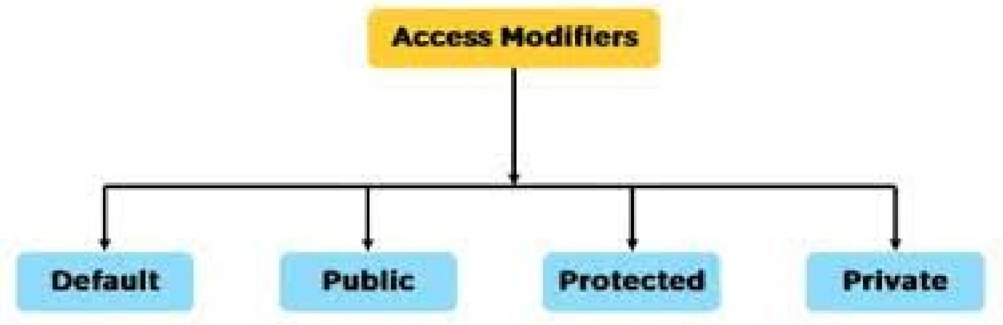
```
# The string representation of an object WITH the __str__() function:
class Person:
    def __init__(self, name, age, address):
        self.name = name
        self.age = age
        self.address = address

    def __str__(self):
        return f"{self.name}({self.age})({self.address})"

p1 = Person("Zahid", 24, "Mumbai")
print(p1)
```

```
Zahid(24)(Mumbai)
```

Access Modifiers



- Access Modifiers (Access Specifiers) are keywords that are used in OOP (object-oriented programming) in order to specify the accessibility of the methods, classes, constructors, and other members of the class.
- In Python, access modifiers are used to control the visibility and accessibility of class attributes and methods.
- They help in encapsulating the internal workings of a class and provide different levels of access to different parts of your code.
- Python has three main access modifier:
    a. Public: Attributes and methods marked as public are accessible from anywhere in your code. They are denoted without any underscores (e.g., public_var, public_method).
    b. Protected: Attributes and methods marked as protected are meant to be accessed within the class itself and its subclasses. They are denoted with a single underscore prefix (e.g., _protected_var, _protected_method).
    c. Private: Attributes and methods marked as private are only accessible within the class they are defined in. They are denoted with a double underscore prefix (e.g., __private_var, __private_method).

```
class myclass:
    def __init__(self):
        self.public_var = "I am a public variable" # All data members
and member functions of a class are public by default.
        self._protected_var = "I am a protected variable"
        self.__private_var = "I am a private variable"

    def public_method(self):
        print("This is a public method.")

    def _protected_method(self):
        print("This is a protected method.")

    def __private_method(self):
        print("This is a private method.")

obj = myclass()

# Accessing public members
print(obj.public_var)
obj.public_method()

# Accessing protected members (though it's a convention to treat it as
protected)
print(obj._protected_var)
obj._protected_method()

# Accessing private members (name mangling changes the variable name)
# However, you can still access it with the mangled name
# like `_MyClass__private_var`
# print(obj.__private_var)  # This will raise an AttributeError
print(obj._myclass__private_var)
obj._myclass__private_method()
```

```
I am a public variable
This is a public method.
I am a protected variable
This is a protected method.
I am a private variable
This is a private method.
```

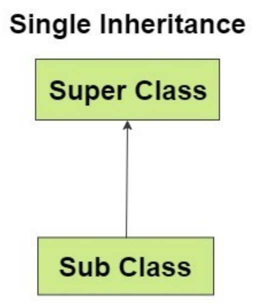
Inheritance

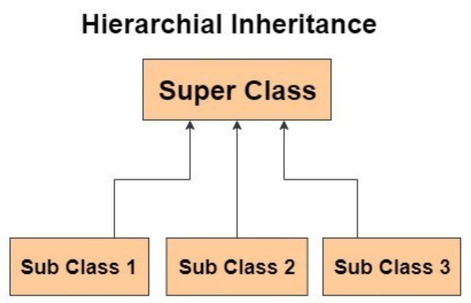
- Inheritance is the capability of one subclass inherits the properties & method from superclass.
- The main idea behind this is to reuse the code in superclass just by inheriting it in subclass which save effort and memory.
- There are 5 types of inheritance:-
    a. Single Inheritance:- One subclass inherites properties & method from superclass.
    b. Multilevel Inheritance:- One class is derived by another class which is also derived from another class. For eg. If there is 3 classes A is the superclass & B & C is the subclass. B inherits properties and method from A and C inherits properties and

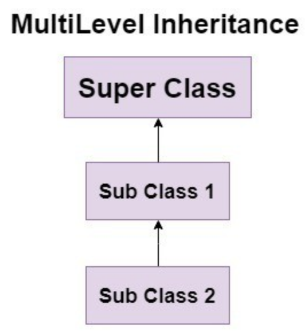
method from B then indirectly C inherits properties and method from A. This is called Multilevel

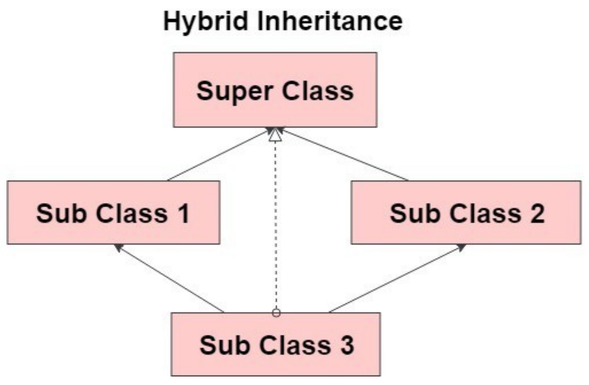
c. Hierachical Inheritance:- Here thre will be only one parent class form which we derived 2 or more child class
d. Multiple Inheritance:- In this subclass inherits properties from 2 or more superclass
e. Hybrid Inheritance:- It is combination of the above types.

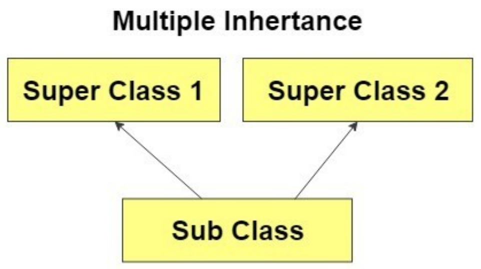
- Constructors & private variable cannot be inherit using Inheritance
- One can derive a child class from parent class and add new features to it without modifying its parents.
- For eg. Father and Son Relationship. In this son inherits father's properties but father cannot get son's properties











```python
# Parent class
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

    def drive(self):
        return f"This {self.brand} vehicle is driving."

# Child class inheriting from Vehicle
class Car(Vehicle):
    def honk(self):
        return f"The {self.brand} car is honking."

# Creating instances
car_instance = Car("Bugatti")

# Using methods from both parent and child classes
print(car_instance.drive())
print(car_instance.honk())
```

```
This Bugatti vehicle is driving.
The Bugatti car is honking.
```

The super() Function

- Python also has a super() function that will make the child class inherit all the methods and properties from its parent
- By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

```
class Vehicle:
    def __init__(self, windows, doors, enginetype):
        self.windows = windows
        self.doors = doors
        self.enginetype = enginetype

class Ferrari(Vehicle):
    def __init__(self, windows, doors, enginetype, enableai):
        super().__init__(windows, doors, enginetype)  # Use super() to 
call the parent class's __init__ method
        self.enableai = enableai

ferrari_laferrari = Ferrari(2, 2, "hybrid", True)

dir(ferrari_laferrari)
```

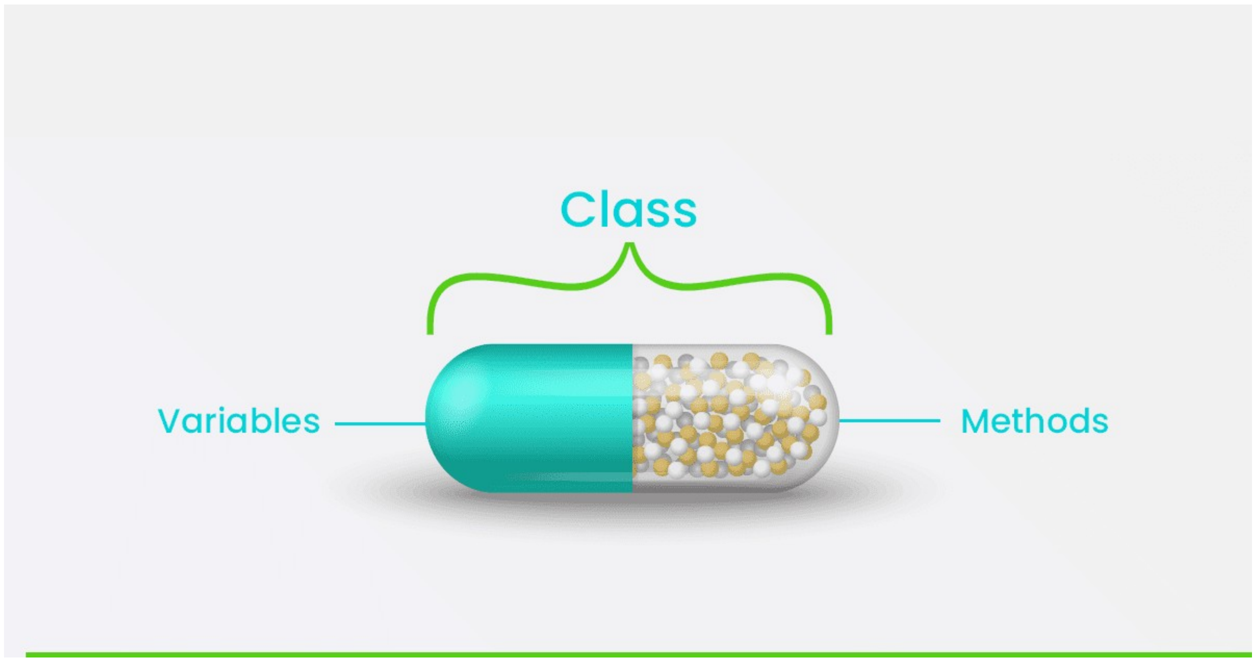
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'doors',
 'enableai',
 'enginetype',
 'windows']
```

```python
print(ferrari_laferrari.windows)
print(ferrari_laferrari.enableai)
```

```
2
True
```

Encapsulation

- It combine the data with properties and method together in form of class.
- Encapsulation in Python is achieved using access specifiers to control the visibility and accessibility of class attributes and methods.
- Access specifiers help in enforcing the principle of information hiding, where the internal details of a class are hidden from the outside world, promoting modularity and reducing the risk of unintended interference.
- Encapsulation is a process of hiding details & protecting data and behavier of the object. It works on Data Hiding.
- Data Hiding means the variable of a class will be hidden from other classes and can only be access through the method of the current class
- For eg. Imagine you have a bank account with sensitive information like account number and balance. You wouldn't want anyone to access this information directly. This is where encapsulation comes in. In the bank account example, encapsulation acts like a security guard protecting your sensitive data.



```
class BankAccount:
    def __init__(self, account_number, name, balance):
        self.__account_number = account_number  # Private attribute,
prefixed with two underscores
        self.__name = name
        self.__balance = balance

    def get_account_number(self):
        return self.__account_number

    def get_name(self):
        return self.__name

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

    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
        else:
            print("Insufficient funds")

    def get_balance(self):
        return self.__balance


# Create a bank account object
account1 = BankAccount("12345", "Zahid Salim Shaikh", 1000000)

name = account1.get_name()
```

```python
print(f"Name: {name}")

# Access account number through getter method
account_number = account1.get_account_number()
print(f"Account number: {account_number}")

# Deposit money
account1.deposit(1000000)


# Cannot directly access balance
# print(account1.__balance)  # This will raise an AttributeError

# Get balance through getter method
current_balance = account1.get_balance()
print(f"Current balance: {current_balance}")

# Withdraw money
account1.withdraw(1200)
```

```
Name: Zahid Salim Shaikh
Account number: 12345
Current balance: 2000000
```

Getter and Setter

- It is used to mdoify and retrieve value of the variable.
- The setter method is used for uodating the values.
- The getter method is used for retrieving the values.

Polymorphism

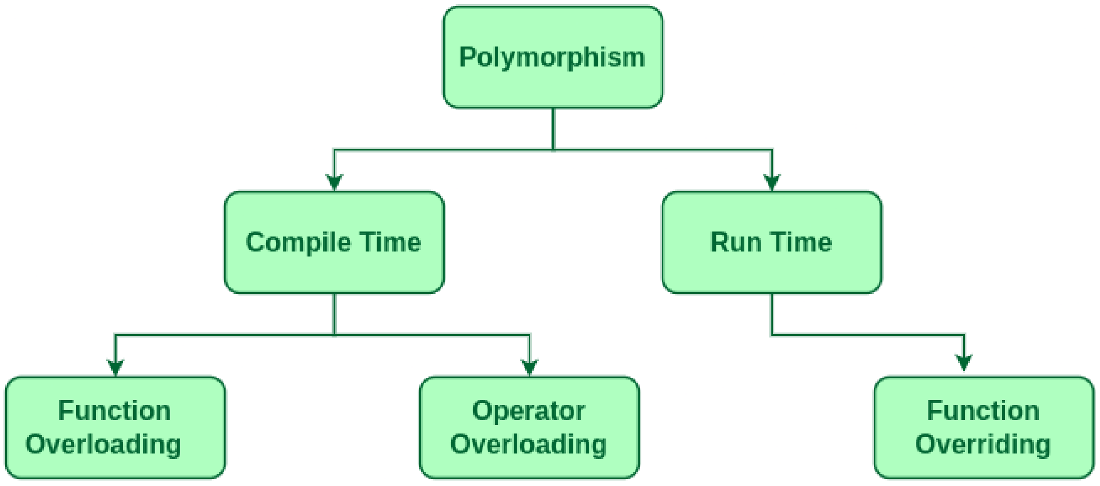
- It allows us to perform single action in different forms.
- There are two types of polymorphism:-
  a. Runtime Polymorphism
  b. Compile time Polymorphism

1. Compile time Polymorphism:
   - It is also called as static Polymorphism.
   - This types of Polymorphism is achieved by method overloading.
   - In this process call to the method is resolved at compile-time
     - Method Overloading:
       - When there is multiple methods with same name and in same class by different parameter then these method is called as method overloading.
       - Function can be over loaded by change number of parameter, sequence of parameter or the types of parameter
     - Method Hiding:

        - It is similar to overriding.
        - A static method cannot be overridden. But a static method is defined in the parent class is redefine in the child class. The child class method hides the method defined in parent class.

2. Runtime Polymorphism:
    - It is also known as dynamic method dispatch.
    - It is a process in which call to the method is resolved at runtime.
    - This type of polymorphism is achieve by method overriding.
        - Method Overriding:
            - When there is multiple method with same name and same parameter but different classes it is known as overloading.
            - It occurs when derived subclass has one of the member functions of the superclass. Then the base function is said to be overwritten.

- For Example:- A person at a same time can have different characters like a man, at a same time is a father, a husband, a son, a friend, an employee. So same person have different behavioues. This is called as Polymorphism.



```
# Base class with a method for calculating the area
class Shape:
    def area(self):
        pass  # Placeholder for subclasses to override

# Subclass that inherits from Shape and calculates the area of a
circle
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

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

# Subclass that inherits from Shape and calculates the area of a
rectangle
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

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

# Create instances of Circle and Rectangle
circle = Circle(5)
rectangle = Rectangle(4, 6)

# Use polymorphism to calculate and print the areas
shapes = [circle, rectangle]

for shape in shapes:
    print("Area:", shape.area())
```

```
Area: 78.5
Area: 24
```

Abstraction

- Abstraction is a fundamental object-oriented programming concept that focuses on hiding the internal details of an object and exposing only its essential functionalities.
- It allows us to deal with objects at a higher level, simplifying our code and making it easier to understand and maintain.
- Abstraction hides complex implementations behind simple interfaces, making code more readable and manageable.
- By focusing on common functionalities, we can create reusable components that can be used in different parts of the program.
- Abstraction makes code easier to update and modify, as changes to internal details can be implemented without affecting the external interface.
- For eg. the Car represents the abstraction of a real-world car. We only expose essential functionalities like starting the engine, accelerating, and stopping the engine through methods. The internal details of how these functions are implemented are hidden from the user, making the code more concise and user-friendly.

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

    def start_engine(self):
```

```python
        print(f"{self.model} is starting its engine.")

    def accelerate(self):
        print(f"{self.model} is accelerating.")

    def brake(self):
        print(f"{self.model} is applying brake.")

    def stop_engine(self):
        print(f"{self.model} is stopping its engine.")

# Create a Car object
car1 = Car("Lamborghini Aventador", "Black")

# Use the car object without worrying about its internal
implementation
car1.start_engine()
car1.accelerate()
car1.brake()
car1.stop_engine()
```

```
Lamborghini Aventador is starting its engine.
Lamborghini Aventador is accelerating.
Lamborghini Aventador is applying brake.
Lamborghini Aventador is stopping its engine.
```