OBJECT ORIENTED PROGRAMMING SYSTEM

1. What is OOPS and why it is needed ?

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into reusable units called objects, which encapsulate data (attributes) and behavior (methods).

Why needed ?

- Modular Code base
- Reusable Code
- Scalable for large application
- Hides Implementation Details (Abstraction)
- Maintainable Code
- Bottom up approach

EX : Java , C++ , Python

2. What are the main features of OOPS ?

OOPS have main 4 key features :

- **Inheritance:** Allowing another class to inherits properties and methods from another
- **Encapsulation:** Binding data into single unit ad restricting the access of to protect it.
- **Polymorphism:** Different forms of behaviours with same name.
- **Data abstraction:** Hiding complex details and showing essential features.

3. What is Class and Object ?

Class :

A Class is a blue print or template for creating objects. It defines the properties (attributes) and behaviour (methods) that its object have

Object :

A object is an instance of a class that represents a real world entity.

4. What is a constructor ?

Constructors are special methods whose name is the same as the class name.

Constructor serve the special purpose of intializing the objects.

5. What are the various types of constructors in C++ ?

Default Constructor: Default constructor is the constructor which doesn't take any argument.

Parameterized Constructor: Constructor that take some arguments are known as parameterized constructors.

Copy Constructor: Copy Constructor is a type of constructor, whose purpose is to copy an object to another.

6. What is a destructor ?

Destructors are also special methods and destructors are automatically called when object is being destroyed.

7. What is Self ?

- Self is not other than it is simple object.
- id(self) = id(object)

8. Why we need Self ?

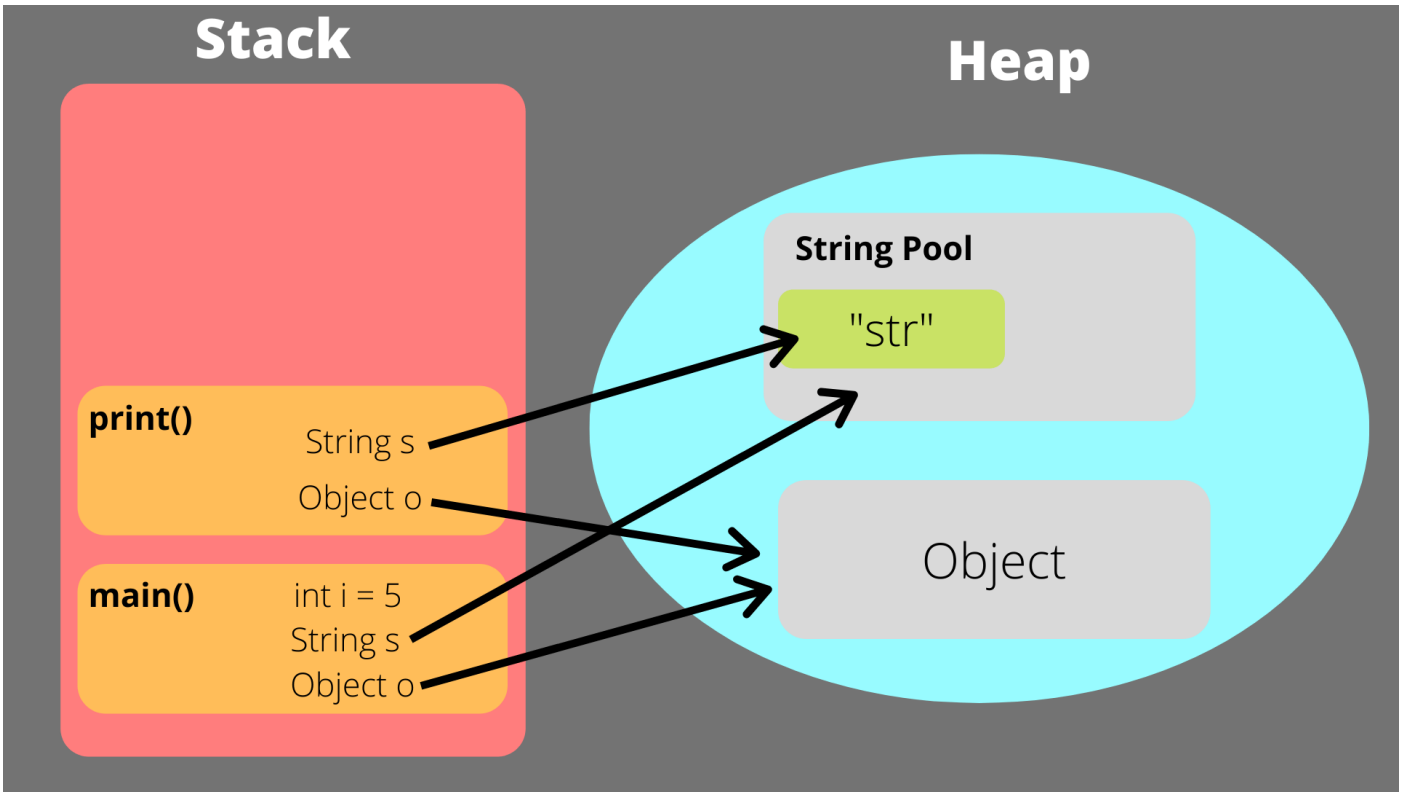
We cannot access any method from another method inside class therefor we need the Self object.

9. What is instance Variable ?

- Variable which are made under the constructor are known as instance variable.
- All The instance variable values are different for different objects.

10. What is difference between structure and class ?

- Structure saved in Stack memory and Data abstraction is not possible in Structure.
- Class is stored in Heap memory and Abstraction is possible in Class.



Feature	Stack Memory	Heap Memory
Management	Automatic (by system)	Manual (by programmer)
Speed	Faster	Slower
Size	Limited and predefined	Larger, can grow dynamically
Lifetime	Tied to function scope	Defined by programmer
Usage	Local variables, function calls	Dynamic allocation
Thread-Safety	Inherently thread-safe	Requires synchronization
Error Risk	Stack overflow	Memory leaks, fragmentation

11. How Much memory does class occupy ?

Class does not consume any memory because It is only blueprint. When objects are created actually it initialized the class and consumes the memory.

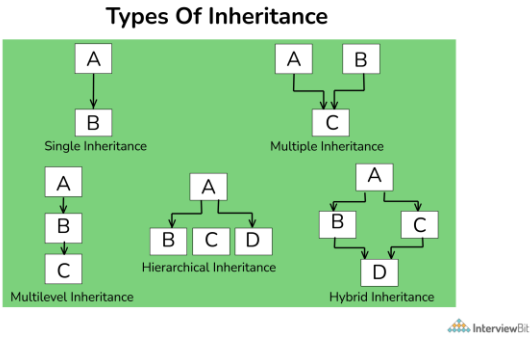
12. Is it always necessary to create objects from class?

- No. An object is necessary to be created if the base class has non-static methods.
- But if the class has static methods, then objects don’t need to be created. You can call the class method directly in this case, using the class name

1. INHERITANCE

1. What is the Inheritance ?

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows one class to inherits the properties and behaviours of another class.



Types of Inheritance:

- Single Inheritance
- Multiple Inheritance
- Multi – Level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

2. What is Subclass ?

The subclass is a part of inheritance. It is also known as the child class.

3. What is Superclass ?

The superclass is an entity, which allows subclasses or child classes to inherit from itself.

4. Example of Inheritance:

```python
# Parent class
class Parent:
    def greet(self):
        print("Hello from Parent!")

# Child class
class Child(Parent):
    def greet_child(self):
        print("Hello from Child!")

# Usage
child = Child()
child.greet()  # Accessing parent method
child.greet_child()
```

5. Are there any limitations of inheritance ?

- Inheritance is a very powerful feature in OOPs, but it has some limitations too.
- Inheritance needs more time to process, as it needs to navigate through multiple classes for its implementation.

6. What is the purpose of Super Keyword in inheritance ?

- Super is used to call the parent class's constructor or methods from the derived class.
- Super() keyword always works inside the class and it is used to call the parent class methods and constructor

```
class Parent:
    def __init__(self):
        self.num = 100

    def parent_method(self):
        print("Inside Parent class method accessing through child
self")

    def parent_method2(self):
        print("Inside Parent class method accessing through super")

class Child(Parent):
    def __init__(self):
        super().__init__()
        self.var = 200

    def show(self):
        print(self.num)
        print(self.var)
        print(self.parent_method())
        print(super().parent_method2())

def main():
    x = Child()
    x.show()

if __name__ == "__main__":
    main()
```

7. What happens if you do not explicitly define a constructor in a derived class ?

- The parents class default constructor is called automatically.

8. Example of Different Type of Inherintance.

1. Single Inheritance

```
# Parent class
class Parent:
    def greet(self):
        print("Hello from Parent!")

# Child class
class Child(Parent):
    def greet_child(self):
        print("Hello from Child!")

# Usage
child = Child()
child.greet()  # Accessing parent method
child.greet_child()
```

2. Hierarchical Inheritance

```
# Parent class
class Parent:
    def greet(self):
        print("Hello from Parent!")

# Child classes
class Child1(Parent):
    def greet_child1(self):
        print("Hello from Child1!")

class Child2(Parent):
    def greet_child2(self):
        print("Hello from Child2!")

# Usage
child1 = Child1()
child1.greet()
child1.greet_child1()

child2 = Child2()
child2.greet()
child2.greet_child2()
```

3. Multi Level Inheritance

```python
# Base class
class Grandparent:
    def greet(self):
        print("Hello from Grandparent!")

# Intermediate class
class Parent(Grandparent):
    def greet_parent(self):
        print("Hello from Parent!")

# Derived class
class Child(Parent):
    def greet_child(self):
        print("Hello from Child!")

# Usage
child = Child()
child.greet()         # From Grandparent
child.greet_parent()  # From Parent
child.greet_child()   # From Child
```

4. Multiple Inheritance

```python
# Parent classes
class Parent1:
    def greet1(self):
        print("Hello from Parent1!")

class Parent2:
    def greet2(self):
        print("Hello from Parent2!")

# Child class
class Child(Parent1, Parent2):
    pass

# Usage
child = Child()
child.greet1()
child.greet2()
```

2. ENCAPSULATION

1. What is Encapsulation ?

- Encapsulation is the process of bundling data (variables) and methods (functions) that operate on that data into a single unit, typically a class,
- while restricting direct access to some of the object's components.
- Using Access Modifier We Can Achieve this

2. Give example of Encapsulation

```python
class Student:
    def __init__(self, name):
        self.__name = name  # Private attribute

    # Getter for name
    def get_name(self):
        return self.__name

    # Setter for name
    def set_name(self, name):
        self.__name = name

# Usage
student = Student("Alice")

# Accessing private attribute through getter
print(student.get_name())  # Output: Alice

# Modifying private attribute through setter
student.set_name("Bob")
print(student.get_name())  # Output: Bob
```

Encapsulation can also be defined in two different ways:

1) Data hiding: Encapsulation is the process of hiding unwanted information, such as restricting access to any member of an object.

2) Data binding: Encapsulation is the process of binding the data members and the methods together as a whole, as a class.

* Encapsulation

⟹ object બનાવીએ તો તે classનું બધા વસ્તુ Access કરી શકે છે but તે ના થાય તે માટે encapsulation નો use થાય છે

* using Access Modifier we can Achieve encapsulation

① PRIVATE

__Var name ⟶ private Variable

Nothing

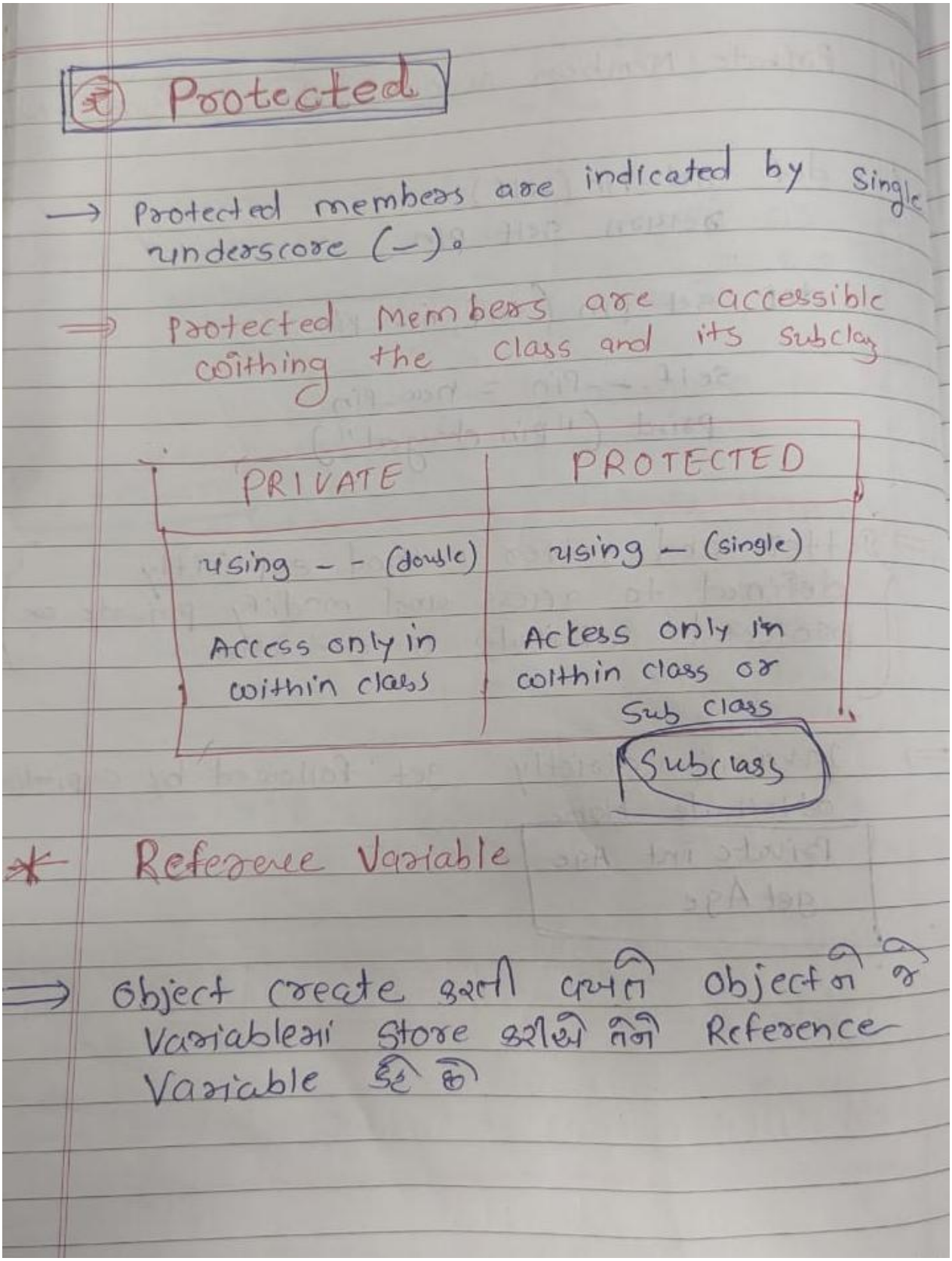
__XYZ ⟶ convert ⟶ _ATM__XYZ

class Name — Var Name

Sbi.__XYZ = "palak" // બધી variable __XYZ બનાવશે

Nothing In Python Truly Private

⟹ Python is language for Adult



What is Access Specifier ?

- Access specifier are a special type of keywords which are used to control the accessibility of entities like classes, methods etc.
- Access specifier play vital role to achieve encapsulation

3. POLYMORPHISM

1. What is Polymorphisms ?

Polymorphism is a concept in programming where a single function, method, or object can take on multiple forms or different behaviours depending on the context.

Polymorphism refers to the process by which some code, data, method, or object behaves differently under different circumstances or contexts.

There Mainly 2 Type of Polymorphism:

1) Method Overloading (Compile Tine Polymorphism / Static Polymorphism)

- Technically Method overloading is not supported in Python But we can achieve this by using default argument

2) Method Overriding (Run Time Polymorphism / Dynamic Polymorphism)

- Runtime polymorphism refers to the type of Polymorphism in OOPs, by which the actual implementation of the function is decided during the runtime or execution.

COMPILE TIME POLYMORPHISM :

```
class Geometry:
    def area(self):
        print("Area method in Geometry class")

    def area(self, a , b=0):
        if b == 0:
            print("Area of square:", a*a)
        else:
            print("Area of rectangle:", a*b)


def main():
    g = Geometry()
    g.area(5)
    g.area(10, 20)
    g.area(7)

if __name__ == "__main__":
    main()
```

Operator overloading (Compile Time Polymorphism)

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        # Overloading the + operator
        return Point(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        # Overloading the - operator
        return Point(self.x - other.x, self.y - other.y)

    def __str__(self):
        # Overloading the string representation
        return f"({self.x}, {self.y})"


# Example usage:
p1 = Point(2, 3)
p2 = Point(4, 1)

result_add = p1 + p2  # Calls the __add__ method
result_sub = p1 - p2  # Calls the __sub__ method

print(f"Addition: {result_add}")  # Output: Addition: (6, 4)
print(f"Subtraction: {result_sub}")  # Output: Subtraction: (-2, 2)
```

OVERLOADING : MULTIPLE IMPLEMENTATIONS (PARAMS DIFFERENT)

OVERRIDING : DEFINE IN DIFFERENT CLASS (SAME PARAMS)

RUN TIME POLYMORPHISM:

```python
# Parent class
class Animal:
    def speak(self):
        return "The animal makes a sound"

# Subclass
class Dog(Animal):
    def speak(self):
        return "The dog barks"

class Cat(Animal):
    def speak(self):
        return "The cat meows"

# Create instances
animal = Animal()
dog = Dog()
cat = Cat()

# Call the speak method
print(animal.speak())  # Output: The animal makes a sound
print(dog.speak())     # Output: The dog barks
print(cat.speak())     # Output: The cat meows
```

4. ABSTRACTION

1. What is Abstraction ?

Abstraction is the process of hiding the complex implementation details of a system and exposing only the essential features or functionality to the user.

In Python,

- **ABC** is the **base class for creating abstract classes**.
- Abstract method is used to enforce the implementation of specific methods in subclasses.
- Use ABC and abstract method when you need to enforce a specific contract (i.e., mandatory methods) across multiple subclasses.
- For example, all shapes should have area() and perimeter() methods, regardless of their type (circle, rectangle, etc.).
- An abstract class serves as a blueprint for other classes.
- you cannot create an instance of an abstract class.

Abstraction is the method of hiding unnecessary details from the necessary ones.

For example, consider a car. You only need to know how to run a car, and not how the wires are connected inside it. This is obtained using Abstraction.

2. What is Abstract Class ?

- An abstract class is a special class containing abstract methods.
- In Abstract class where abstract methods only declare not implemented.
- When Subclass inherits the abstract class and need to use its abstract method they need to define and implement them.

Real Time Example of Abstraction:

When you use an ATM, you interact with a simple interface:

- Insert your card. , Enter your PIN , checking balance and other options

You don’t see the underlying processes:

- Card validation, Communicating with the bank’s server, Updating account balances.

```python
from abc import ABC, abstractmethod

class Shape(ABC):  # Abstract class
    @abstractmethod
    def area(self):
        pass  # Abstract method

    @abstractmethod
    def perimeter(self):
        pass  # Abstract method

    # Concrete method
    def description(self):
        return "This is a shape"

class Rectangle(Shape):  # Concrete class
    def __init__(self, width, height):
        print("Calling Abstract class method using super()")
        print(super().description())
        self.width = width
        self.height = height

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

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

    # Overriding the concrete method
    def description(self):
        return f"This is a rectangle with width {self.width} and height
{self.height}"

# Usage
rect = Rectangle(10, 20)
# shape = Shape()
print(rect.description())          # Output: This is a rectangle with width 10
and height 20
print(f"Area: {rect.area()}")      # Output: Area: 200
print(f"Perimeter: {rect.perimeter()}")  # Output: Perimeter: 60
```

INTERFACE :

1. What is Interface ?

- An interface refers to a special type of class, which contains methods, but not their definition. Only the declaration of methods is allowed inside an interface.
- A class that contains only abstract methods. It can't be instantiated, but other classes can inherit from it.
- To use an interface, you cannot create objects. Instead, you need to implement that interface and define the methods for their implementation.

```python
from abc import ABC, abstractmethod

class Drawable(ABC):  # Interface
    @abstractmethod
    def draw(self):
        pass  # Abstract method, must be implemented

class Circle(Drawable):  # Implements the interface
    def draw(self):
        return "Drawing a circle"

class Square(Drawable):  # Implements the interface
    def draw(self):
        return "Drawing a square"

# Usage
shapes = [Circle(), Square()]
for shape in shapes:
    print(shape.draw())
```

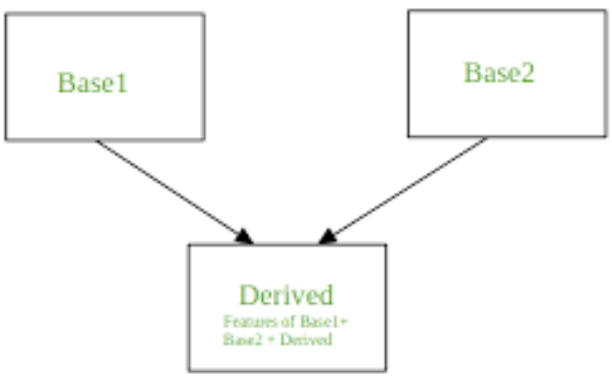
Difference between Interface & Abstract Class

Feature	Interface	Abstract Class
Method Implementation	Only method declarations, no implementation	Can have both abstract and concrete methods
Multiple Inheritance	A class can implement multiple interfaces	A class can inherit from only one abstract class
Constructor	Cannot have constructors	Can have constructors
Access Modifiers	Methods are public by default	Methods can have any access modifier
Fields/Properties	No instance fields, only constants	Can have instance fields
Purpose	To define a contract or behavior	To provide common functionality with some abstract methods
Inheritance	Can implement multiple interfaces	Can inherit from only one abstract class

SHARED CODE : ABSTRACT CLASS

CREATE CONTRACT (METHOD SIGNATURE ONLY) : INTERFACE

MULTIPLE IN-HERITANCE : INTERFACE



MISCELLANEOUS

1. What is Static Method and Static Variable ?

Static Variable:

- Static Variable only one time initialized.
- Always declared outside __init__ method.
- Access using Class Name .

Static Method :

- Static method always without self params.
- Using @staticmethod we can make any method to static.

```
class MyClass:
    # Static variable
    count = 0

    def __init__(self, name):
        self.name = name
        MyClass.count += 1  # Increment count for each new instance

    # Static method
    @staticmethod
    def greet():
        print("Hello! This is a static method.")

    # Instance method to show static variable
    def show_count(self):
        print(f"Total instances: {MyClass.count}")

# Usage
obj1 = MyClass("Alice")
obj2 = MyClass("Bob")

# Access static method
MyClass.greet()  # Output: Hello! This is a static method.

# Access static variable through instance
obj1.show_count()  # Output: Total instances: 2

# Access static variable through class
print(MyClass.count)  # Output: 2\
```

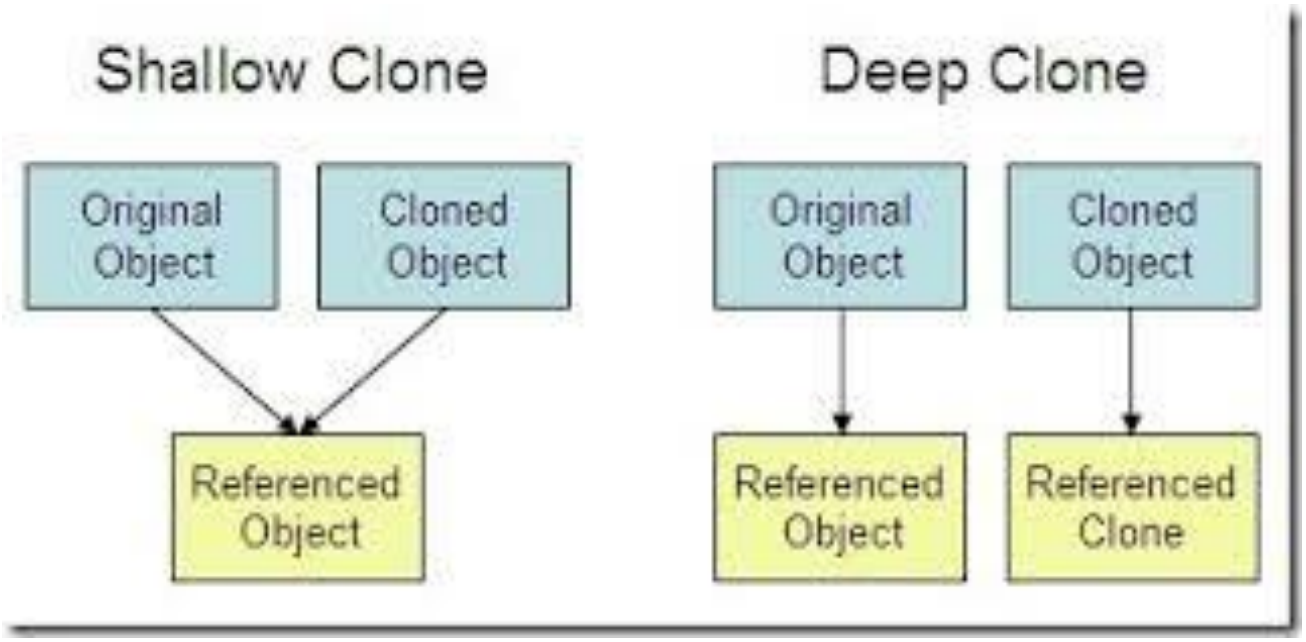
2. What is Shallow Copy and Deep Copy ?

Shallow Copy:

- If the original object contains any references to mutable objects, the copy will also reference the same object.

Deep Copy:

- original object and copied object are completely independent of each other



```python
import copy

print("_____SHALLOW COPY_____")
# Shallow Copy (If the original object contains any references to mutable objects, the copy will also reference the same object.)

original = {"name": "Alice", "hobbies": ["reading", "cycling"]}

shallow_copy = copy.copy(original)

shallow_copy["hobbies"][0] = "swimming"

print("Original List : ", original)  # Output: Original List :  {'name': 'Alice', 'hobbies': ['swimming', 'cycling']}
print("Shallow Copy : ", shallow_copy)  # Output: Shallow Copy :  {'name': 'Alice', 'hobbies': ['swimming', 'cycling']}

# Deep Copy  "(original object and copied object are completely independent of each other)"
print("_____DEEP COPY_____")

original_deep = {"name": "Alice", "hobbies": ["reading", "cycling"]}

deep_copy = copy.deepcopy(original_deep)

deep_copy["hobbies"][0] = "swimming"

print("Original List : ", original_deep)  # Output: Original List :  {'name': 'Alice', 'hobbies': ['reading', 'cycling']}
print("Deep Copy : ", deep_copy)  # Output: Deep Copy :  {'name': 'Alice', 'hobbies': ['swimming', 'cycling']}
```