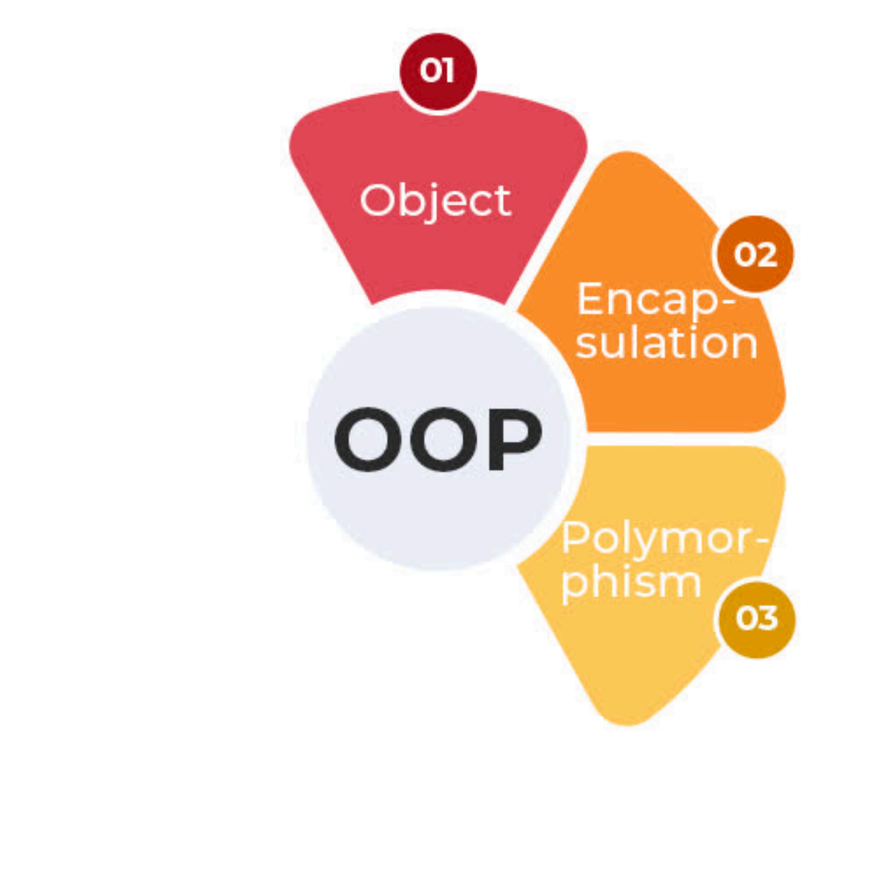
OOP

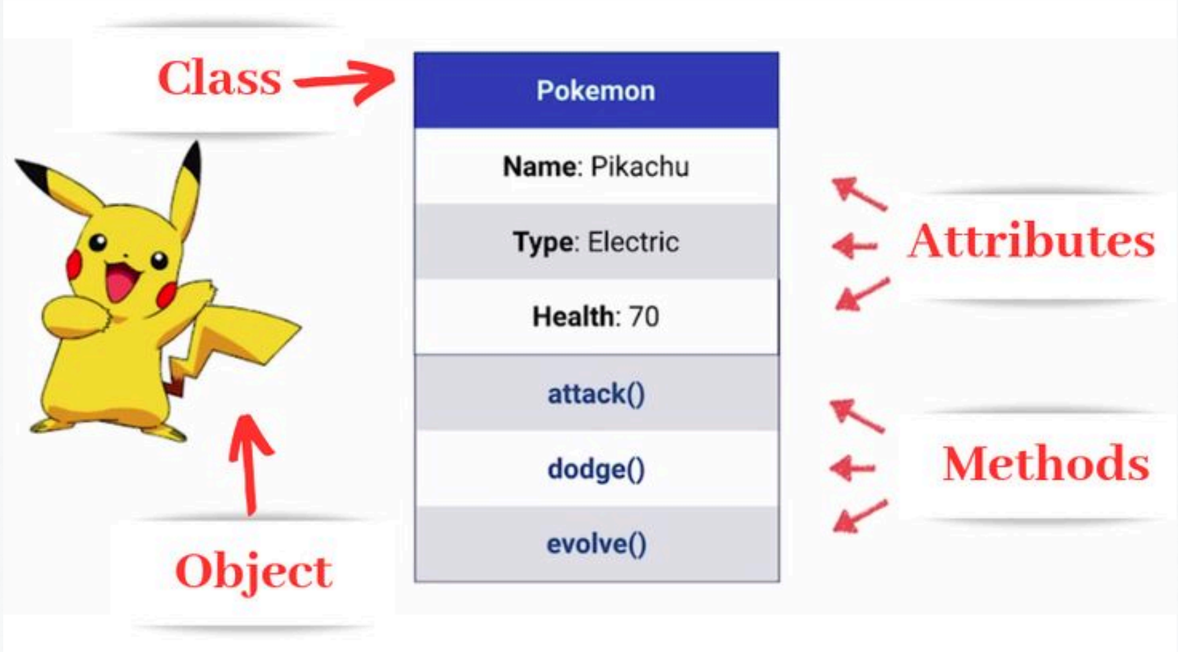
Object-Oriented Programming (OOP) is a programming paradigm that organizes data and behavior into objects.



OOP

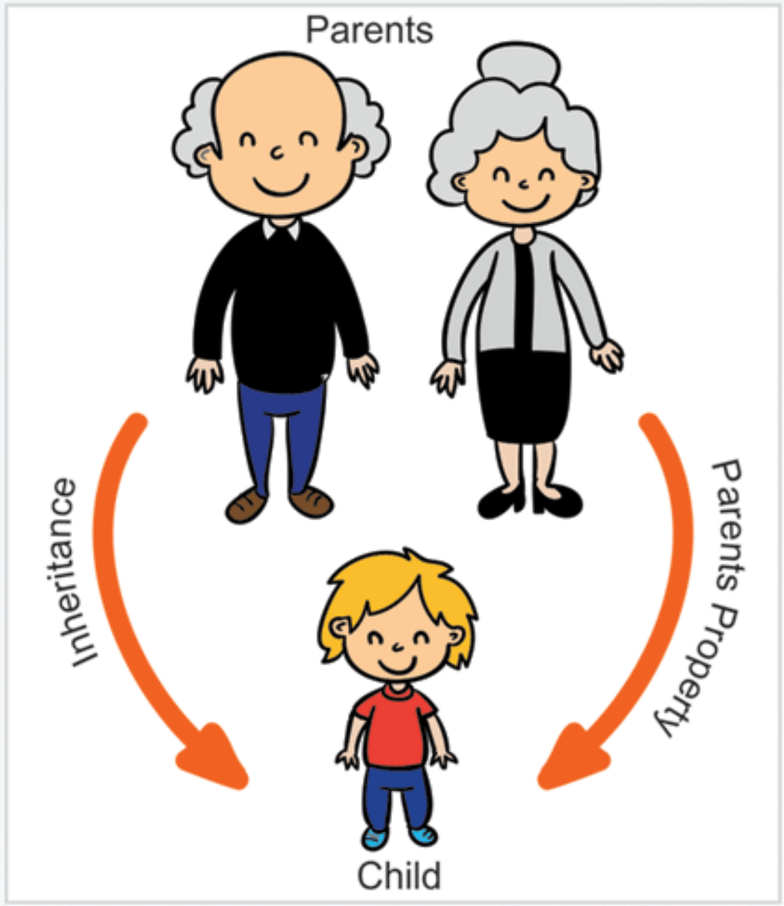
An object consists of:

- **State:** It is represented by the attributes and reflects the properties of an object.
- **Behavior**: It is represented by the methods of an object and reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.



Inheritance

Inheritance allows a class (child) to acquire properties and behaviors of another class (parent). It promotes code reusability.



Inheritance

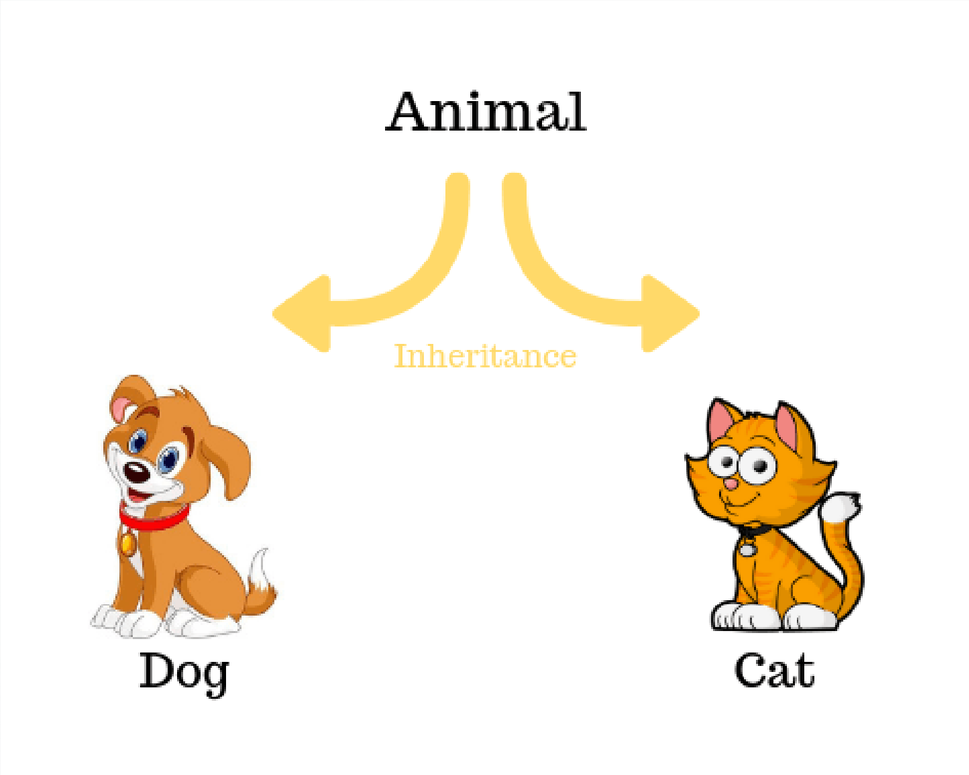
Inheritance allows a class (child) to acquire properties and behaviors of another class (parent). It promotes code reusability.

- **Single Inheritance**: A subclass inherits from one superclass.
- **Multiple Inheritance**: A subclass inherits from more than one superclass.
- **Multilevel Inheritance**: A subclass is derived from another subclass.
- **Hierarchical Inheritance**: Multiple subclasses inherit from a single superclass.
- **Hybrid Inheritance**: A combination of two or more types of inheritance.

Single Inheritance

A subclass inherits from one superclass.

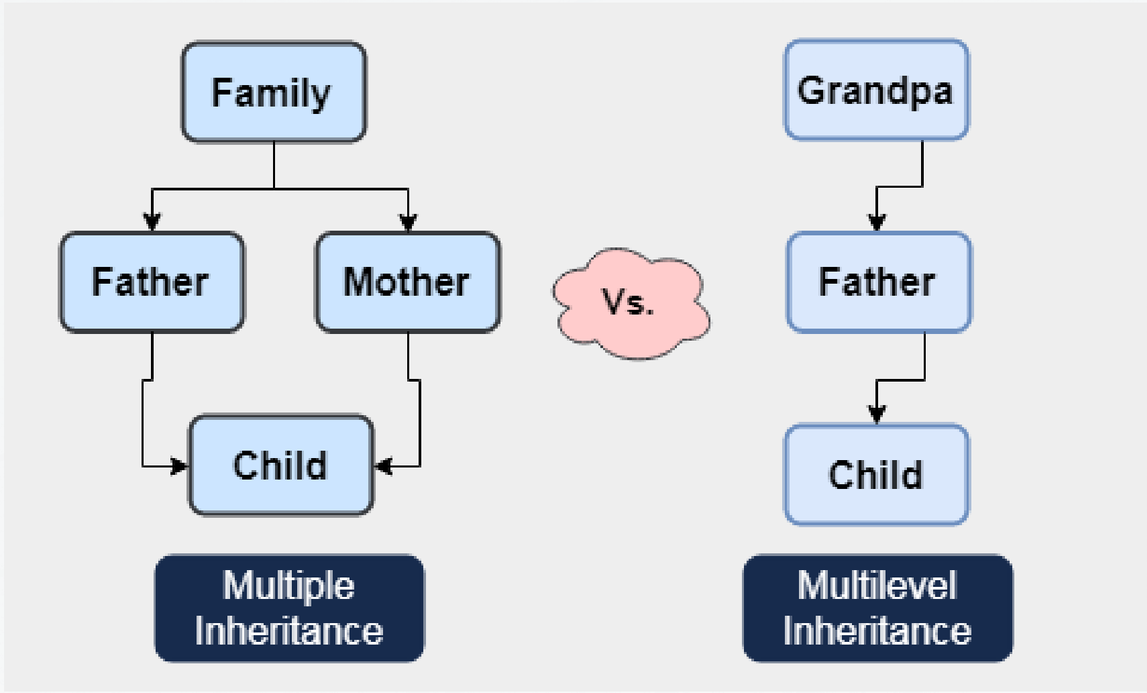
```python
class Animal:
    def sound(self):
        print("Animal makes sound")

class Dog(Animal):  # Dog inherits from Animal
    def bark(self):
        print("Dog barks")

dog = Dog()
dog.sound()  # Inherited method
dog.bark()   # Dog's own method
```



Multiple Inheritance

In multiple inheritance, a subclass inherits from more than one superclass.

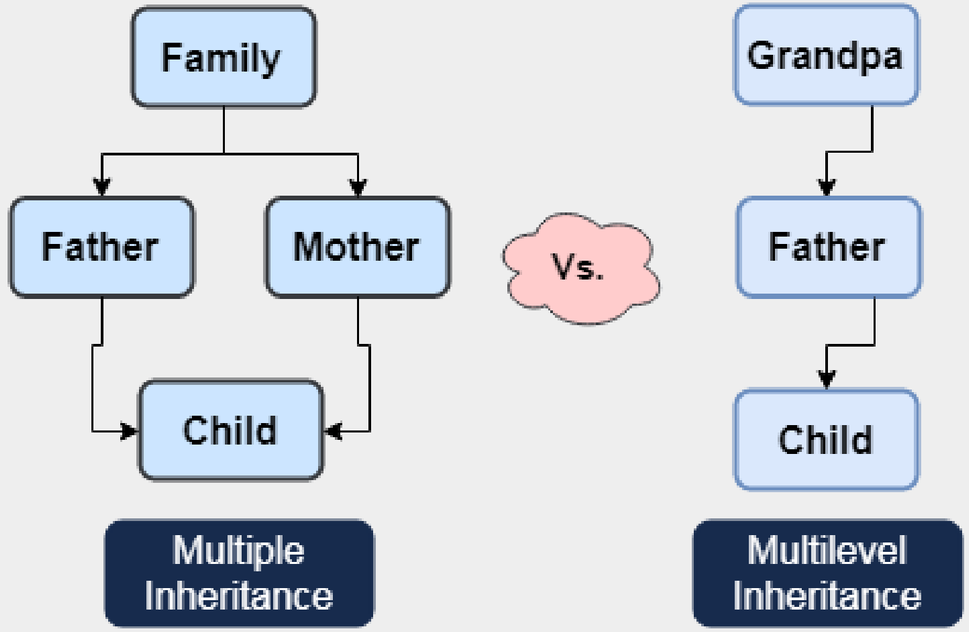
```python
# Parent 1
class Father:
    def height(self):
        return "I am tall like my father."

# Parent 2
class Mother:
    def skin_color(self):
        return "I have fair skin like my mother."

# Child inheriting from both Father and Mother
class Child(Father, Mother):
    def hobby(self):
        return "I love playing football."

child = Child()
print(child.height())      # Output: I am tall like my father.
print(child.skin_color())  # Output: I have fair skin like my mother.
print(child.hobby())       # Output: I love playing football.
```



Multilevel Inheritance

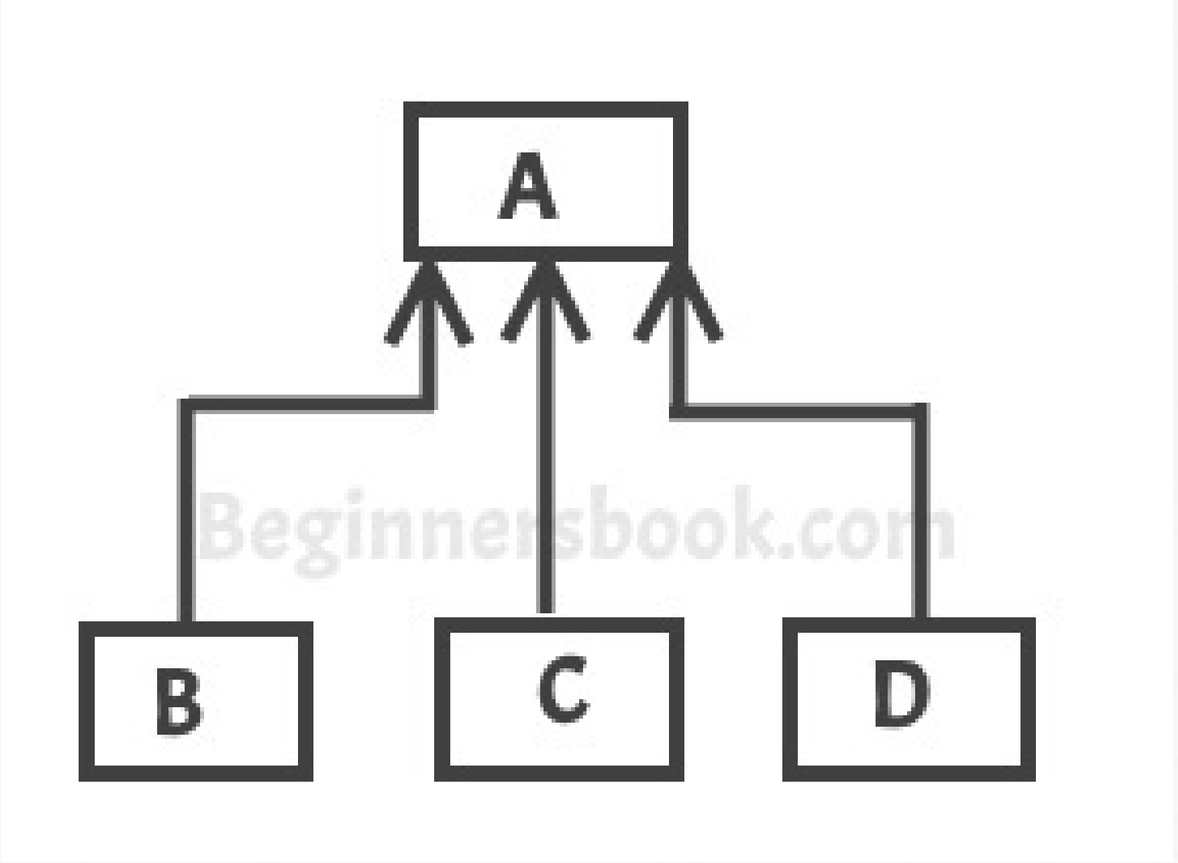
a class inherits from another class, which then inherits from another class, and so on

```python
# Grandparent class
class Grandpa:
    def wisdom(self):
        return "I have a lot of experience in life."

# Parent class (inherits from Grandpa)
class Father(Grandpa):
    def profession(self):
        return "I am a doctor."

# Child class (inherits from Father)
class Child(Father):
    def hobby(self):
        return "I love painting."

child = Child()
print(child.wisdom())      # Output: I have a lot of experience in life.
print(child.profession())  # Output: I am a doctor.
print(child.hobby())       # Output: I love painting.
```



Hierarchical Inheritance

a single class (the superclass) is inherited by multiple classes (the subclasses)

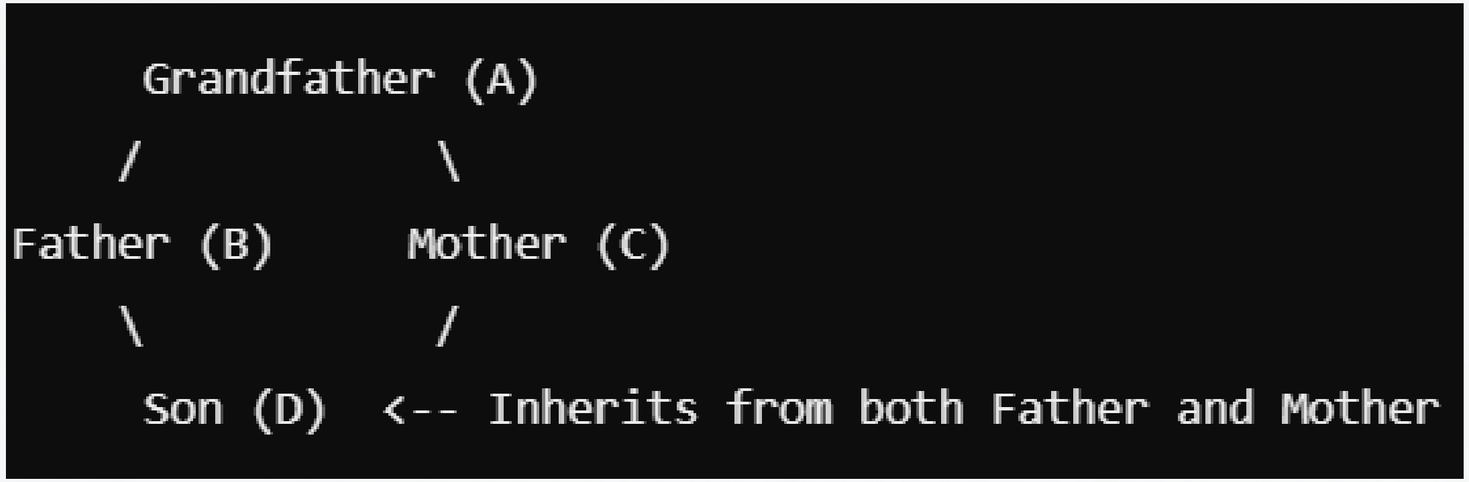


Hierarchical Inheritance

```python
# Parent Class (Father)
class Father:
    def surname(self):
        return "Smith"

    def family_business(self):
        return "Runs a Construction Company"

# Child Class 1 (Son)
class Son(Father):
    def profession(self):
        return "Software Engineer"

# Child Class 2 (Daughter)
class Daughter(Father):
    def hobby(self):
        return "Loves Painting"

# Creating instances of Son and Daughter
son = Son()
daughter = Daughter()

# Accessing parent class methods
print("Son's Surname:", son.surname())   # Output: Smith
print("Son's Profession:", son.profession())   # Output: Software Engineer
print("Son's Family Business:", son.family_business())  # Output: Runs a Construction Company

print("\nDaughter's Surname:", daughter.surname())  # Output: Smith
print("Daughter's Hobby:", daughter.hobby())  # Output: Loves Painting
print("Daughter's Family Business:", daughter.family_business())  # Output: Runs a Construction Company
```

Hybrid Inheritance

Hybrid Inheritance is a combination of multiple inheritance types, such as Multilevel, Multiple, and Hierarchical Inheritance.



Hybrid Inheritance

```python
# Parent Class (Grandfather)
class Grandfather:
    def legacy(self):
        return "I built a real estate empire."
# Father Class (inherits from Grandfather)
class Father(Grandfather):
    def profession(self):
        return "I am a Doctor."
# Mother Class (inherits from Grandfather)
class Mother(Grandfather):
    def talent(self):
        return "I am a great musician."
# Son Class (inherits from both Father and Mother → Multiple Inheritance)
class Son(Father, Mother):
    def hobby(self):
        return "I love coding."
# Creating an instance of Son
son = Son()
# Accessing properties from different levels of inheritance
print(son.legacy())     # From Grandfather → Output: I built a real estate empire.
print(son.profession()) # From Father → Output: I am a Doctor.
print(son.talent())     # From Mother → Output: I am a great musician.
print(son.hobby())      # Own method → Output: I love coding.
```

Polymorphism

Polymorphism means the same method name can have different implementations in different classes.

```python
class Dog:
    def make_sound(self):
        return "Bark!"

class Cat:
    def make_sound(self):
        return "Meow!"

# Function using Polymorphism
def animal_sound(animal):
    return animal.make_sound()

dog = Dog()
cat = Cat()

print(animal_sound(dog))  # Output: Bark!
print(animal_sound(cat))  # Output: Meow!
```

Method Overloading

Python does not support true method overloading like Java/C++, but we can achieve it using default arguments or *args.

```python
class MathOperations:
    def add(self, a, b, c=0):  # Default parameter used for overloading effect
        return a + b + c

math = MathOperations()
print(math.add(10, 20))      # Output: 30
print(math.add(10, 20, 30))  # Output: 60
```

Method Overriding

When a child class provides a specific implementation of a method that already exists in the parent class

```python
class Dog:
    def make_sound(self):
        return "Bark!"

class Cat:
    def make_sound(self):
        return "Meow!"

# Function using Polymorphism
def animal_sound(animal):
    return animal.make_sound()

dog = Dog()
cat = Cat()

print(animal_sound(dog))  # Output: Bark!
print(animal_sound(cat))  # Output: Meow!
```

Encapsulation

Encapsulation is hiding data inside a class to prevent direct modification.
We use private variables (__variable) for this.

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

    def deposit(self, amount):
        self.__balance += amount
        return f"Deposited {amount}. New Balance: {self.__balance}"

    def get_balance(self):
        return self.__balance  # Getter method

account = BankAccount(5000)
print(account.get_balance())  # Output: 5000
print(account.deposit(2000))  # Output: Deposited 2000. New Balance: 7000
print(account.__balance)  # AttributeError: 'BankAccount' object has no attribute '__balance'
```

Encapsulation

Access Modifiers

Modifier	Syntax	Access Level
Public	variable	Accessible everywhere
Protected	_variable	Accessible within the class and subclasses
Private	__variable	Accessible only inside the class

Abstract Classes vs Interfaces

Abstract Class

- Cannot be instantiated (you cannot create an object from it).
- Can have both abstract (no implementation) and concrete (implemented) methods.
- Use ABC (Abstract Base Class) and @abstractmethod decorator.

Interface (Python Alternative)

Python doesn't have built-in interfaces like Java/C#. However, we can achieve interface-like behavior by:

✓ Using abstract classes with only abstract methods

✓ Enforcing subclass implementation without any default behavior

Abstract Classes

```python
from abc import ABC, abstractmethod
# Abstract class
class Vehicle(ABC):
    @abstractmethod
    def start(self):
        pass  # Abstract method (no implementation)

    def fuel_type(self):
        return "Petrol or Diesel"  # Concrete method (implemented)

# Concrete class implementing abstract methods
class Car(Vehicle):
    def start(self):
        return "Car starts with a key."

class Bike(Vehicle):
    def start(self):
        return "Bike starts with a button."
# Instantiating subclasses
car = Car()
bike = Bike()
print(car.start())      # Output: Car starts with a key.
print(bike.start())     # Output: Bike starts with a button.
print(car.fuel_type())  # Output: Petrol or Diesel (Concrete method from Vehicle)
```

Interface

```python
from abc import ABC, abstractmethod

# Interface (abstract class with only abstract methods)
class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass  # No implementation, acts as an interface

class Dog(Animal):
    def make_sound(self):
        return "Bark!"

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

# Instantiating subclasses
dog = Dog()
cat = Cat()

print(dog.make_sound())  # Output: Bark!
print(cat.make_sound())  # Output: Meow!
```

Design Pattern

Singleton Pattern
Ensures that only one instance of a class exists.

```python
class Singleton:
    _instance = None  # Class-level variable

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

s1 = Singleton()
s2 = Singleton()

print(s1 is s2)  # Output: True (Both refer to the same instance)
```

Design Pattern

Factory Pattern

- **Creates objects without specifying the exact class.**

```python
class Circle:
    def draw(self):
        return "Drawing Circle"

class Square:
    def draw(self):
        return "Drawing Square"

class ShapeFactory:
    @staticmethod
    def get_shape(shape_type):
        if shape_type == "Circle":
            return Circle()
        elif shape_type == "Square":
            return Square()
        else:
            return None

shape1 = ShapeFactory.get_shape("Circle")
print(shape1.draw())  # Output: Drawing Circle
```

Design Pattern

Builder Pattern

Helps in creating complex objects step by step.

```python
class Burger:
    def __init__(self):
        self.ingredients = []

    def add_ingredient(self, ingredient):
        self.ingredients.append(ingredient)
        return self  # Returning self allows method chaining

    def build(self):
        return f"Burger with {' '.join(self.ingredients)}"

# Using the Builder Pattern
burger = Burger().add_ingredient("Lettuce").add_ingredient("Tomato").add_ingredient("Cheese")
print(burger)  # Output: Burger with Lettuce Tomato Cheese
```