



# Class, Object & Method

**By Aksadur Rahman**

**[aksadur@yahoo.com](mailto:aksadur@yahoo.com)**

# Agenda

Object Oriented Programing (OOP)

Classes

Objects

Introducing Method

Default Constructors

Parameterized Constructor

Pass Statement

Intro to Inheritance

Single Inheritance

Hierarchical Inheritance

Multilevel Inheritance

Multiple Inheritance

Method Overloading

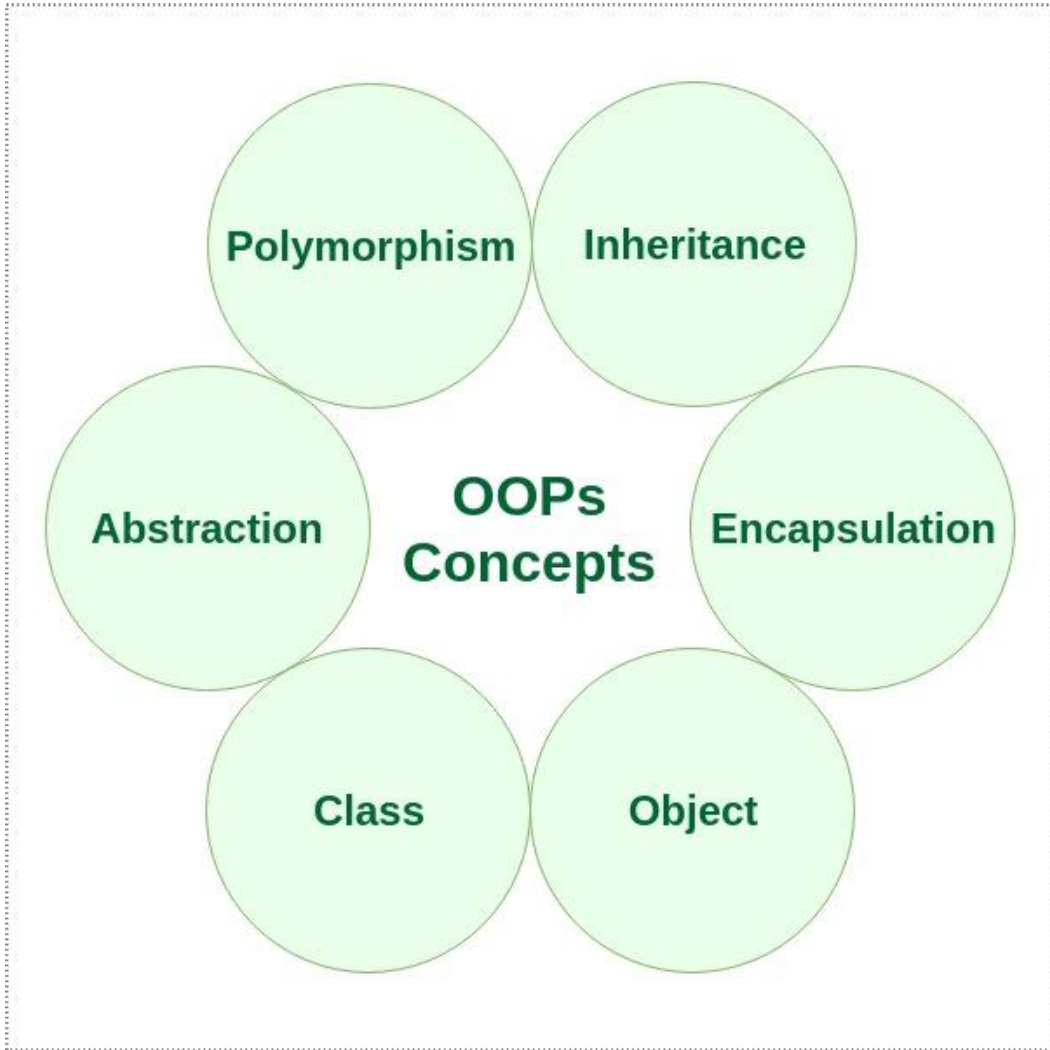
Method Overriding

Encapsulation

Polymorphism

# Object Oriented Programing (OOP)

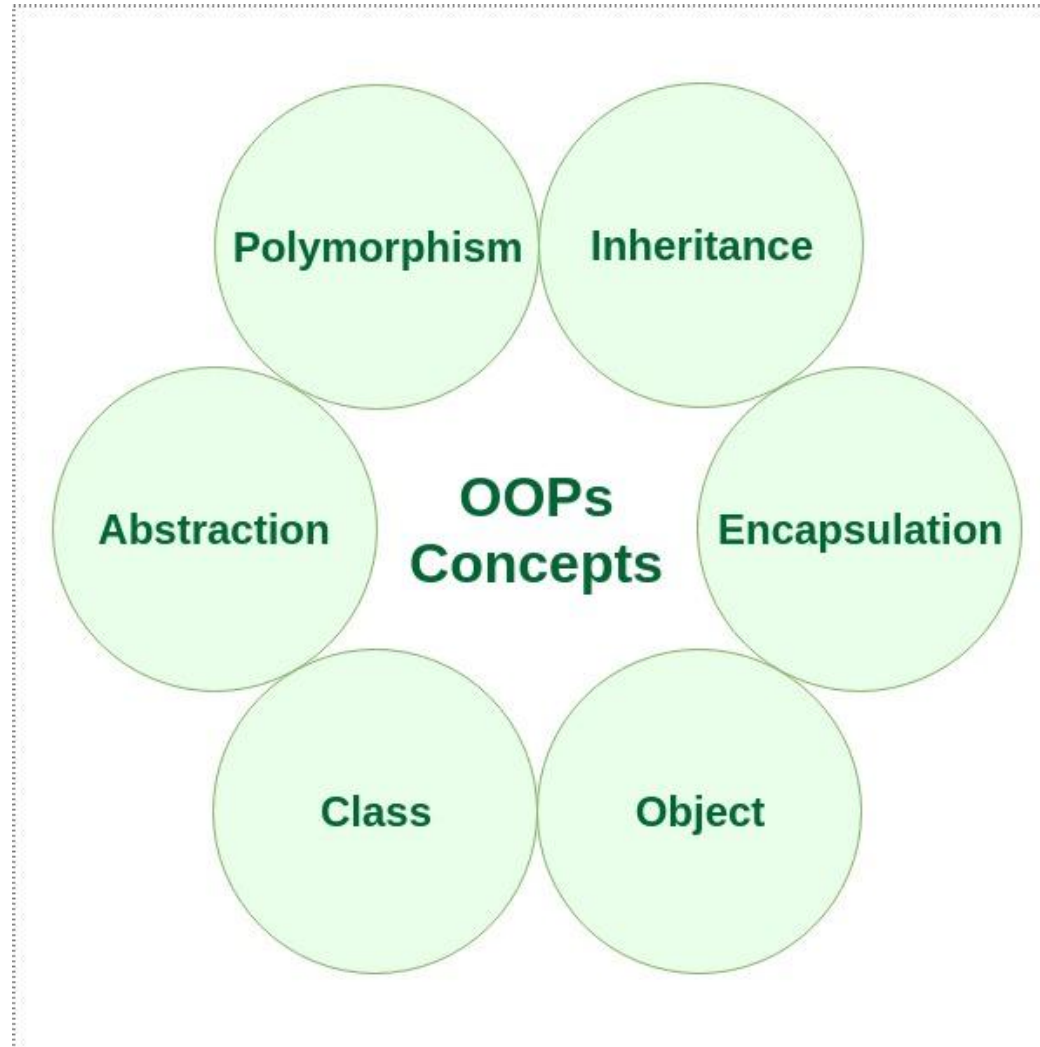
Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.



The diagram illustrates the core concepts of Object-Oriented Programming (OOP). It features a central text 'OOPs Concepts' surrounded by five light green circles, each containing a concept: Polymorphism, Inheritance, Encapsulation, Object, and Class. Abstraction is also present but not in a circle. The entire diagram is enclosed in a dashed rectangular border.

```
graph TD; P((Polymorphism)) --- I((Inheritance)); I --- E((Encapsulation)); E --- O((Object)); O --- C((Class)); C --- A((Abstraction)); A --- P; C --- I; O --- E; P --- OOPs[OOPs Concepts]; I --- OOPs; E --- OOPs; O --- OOPs; C --- OOPs; A --- OOPs
```

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.



# Classes

Classes are user-defined data types that act as the blueprint for individual objects, attributes and methods

An example of a class is the class Student. Students usually have a roll and gpa; these are attributes.

```
class student :  
    roll= ""  
    gpa = ""
```

# Objects

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values.

```
kamal = student()  
kamal.roll = 10  
kamal.gpa = 3.75  
print(f"Roll ={kamal.roll}, GPA={kamal.gpa}")
```

# Introducing Method

A method is a function that “belongs to” an object.

```
class student:
    def set_value(self, a, b):
        self.roll = a
        self.gpa = b

    def display(self):
        print(f"Roll ={self.roll}, GPA={self.gpa}")

kamal = student()

kamal.set_value(10, 3.75)
kamal.display()
```

# Default Constructors

Constructors are generally used for instantiating an object

```
class student:  
    def __init__(self):  
        self.section="A"  
  
    def display(self):  
        print(f"section = {self.section}")  
  
kamal = student()  
kamal.display()
```

# Parameterized Constructors

Constructors are generally used for instantiating an object

```
class student:  
    def __init__(self, roll, gpa):  
        self.roll=roll  
        self.gpa=gpa  
  
    def display(self):  
        print(f"Roll ={self.roll}, GPA={self.gpa}")  
  
kamal = student(10, 3.75)  
kamal.display()
```



# Pass Statement

Create a placeholder for future code:

```
class Person:  
    pass
```

```
def myfunction():  
    pass
```

# Intro to Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

**Parent class** is the class being inherited from, also called base class.

**Child class** is the class that inherits from another class, also called derived class.

## Parent class

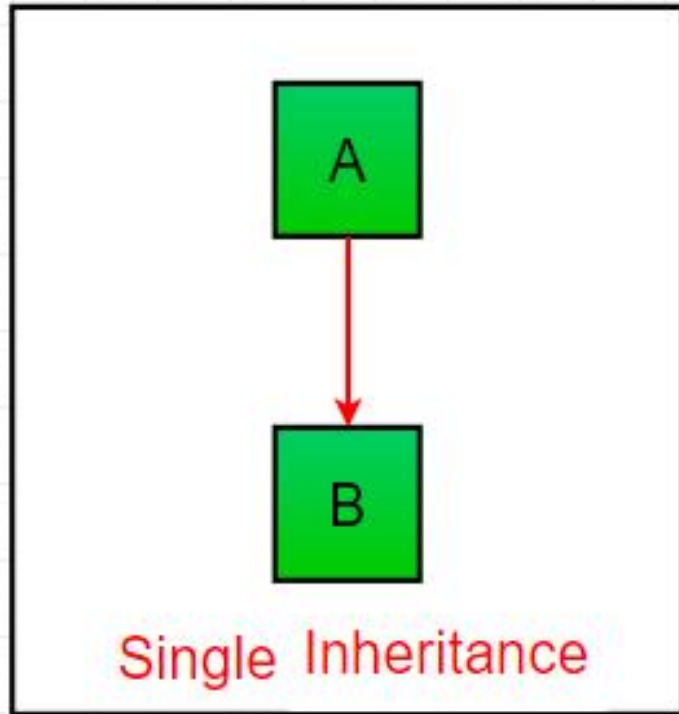
```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
```

## Child class

```
class Student(Person):
    pass
#-----
y = Student("Abul", "Hossain")
y.printname()
```

# Single Inheritance

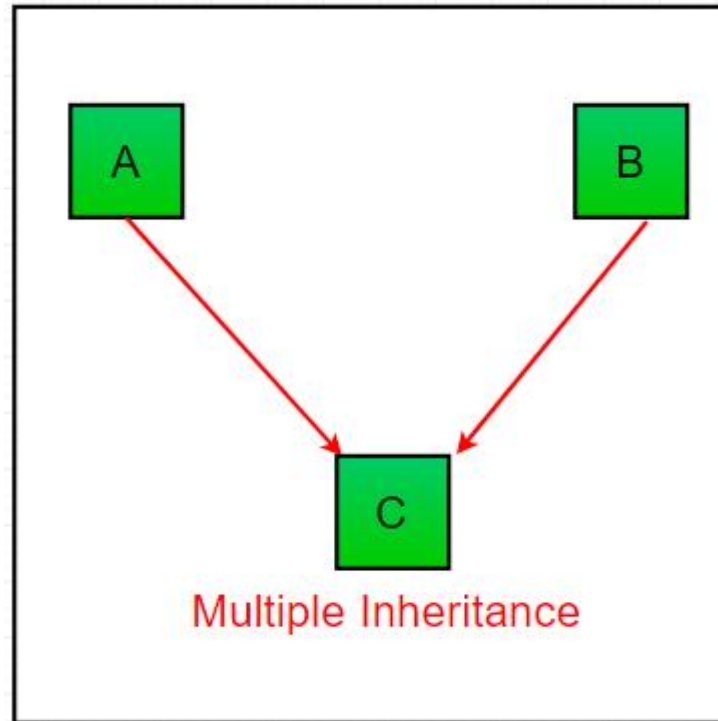


```
class A:
    def display1(self):
        print("This is class A")

class B(A):
    def display2(self):
        print("This is class B")

objB = B()
objB.display1()
objB.display2()
```

# Multiple Inheritance



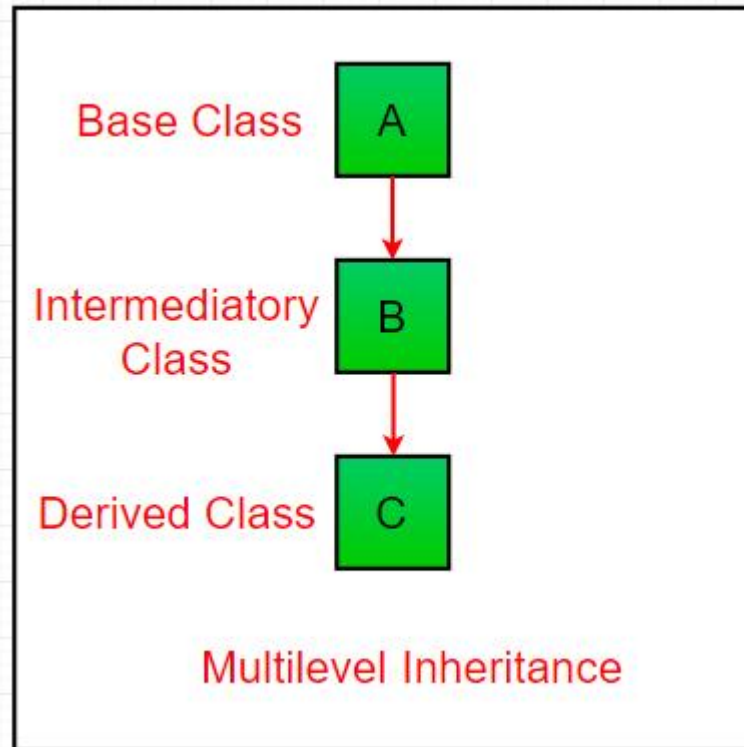
```
class A:  
    def display1(self):  
        print("This is class A")
```

```
class B:  
    def display2(self):  
        print("This is class B")
```

```
class C(A, B):  
    def display3(self):  
        print("This is class C")
```

```
objC = C()  
objC.display1()  
objC.display2()  
objC.display3()
```

# Multilevel Inheritance



```
class A:  
    def display1(self):  
        print("This is class A")
```

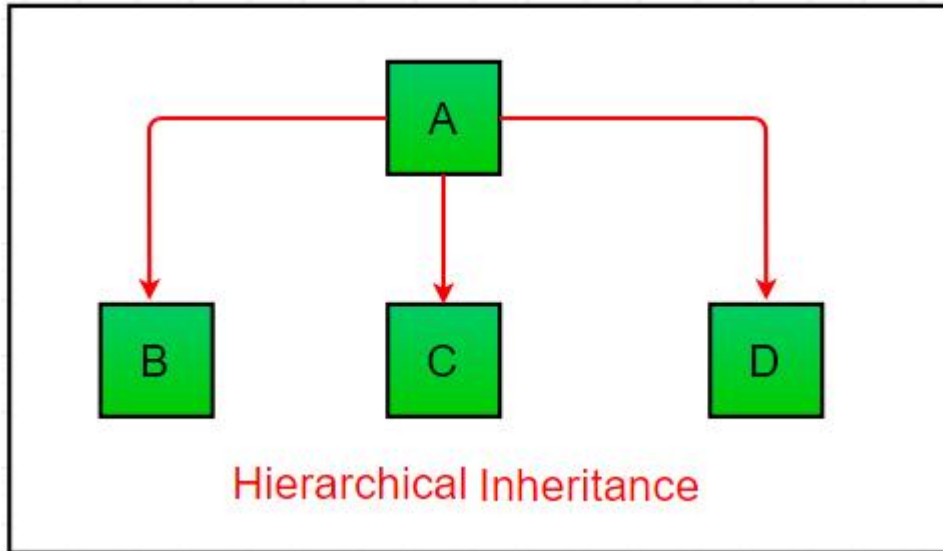
```
class B(A):  
    def display2(self):  
        print("This is class B")
```

```
class C(B):  
    def display3(self):  
        print("This is class C")
```

```
objC = C()
```

```
objC.display1()  
objC.display2()  
objC.display3()
```

# Hierarchical Inheritance



```
class Parent: # Base class
    def func1(self):
        print("This function is in parent class.")
```

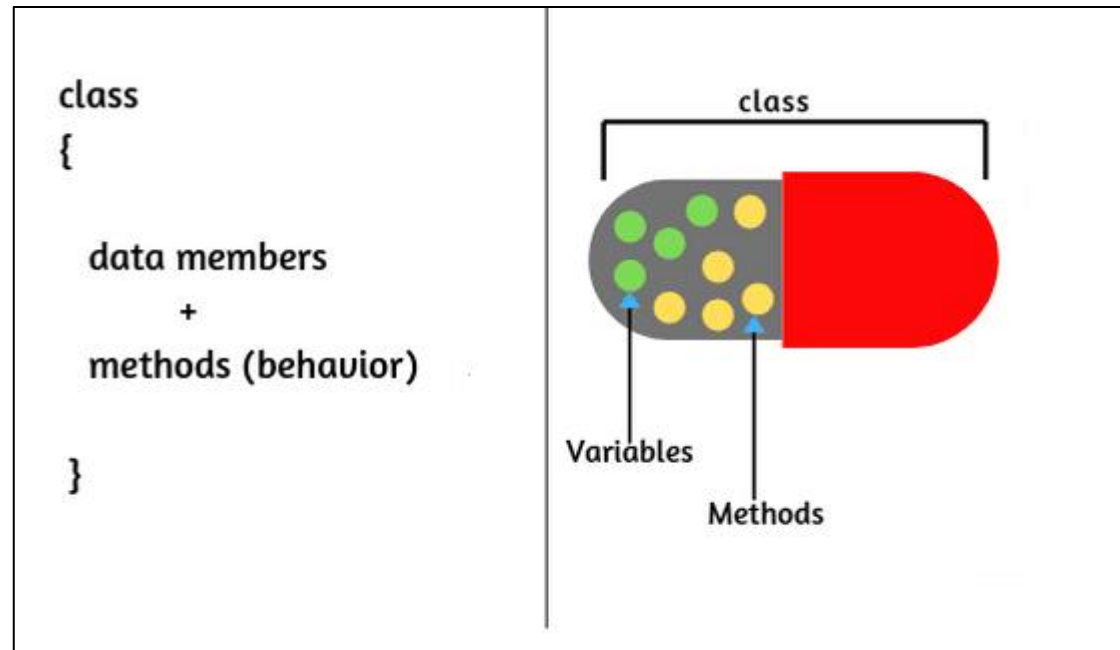
```
class Child1(Parent): # Derived class1
    def func2(self):
        print("This function is in child 1.")
```

```
class Child2(Parent): # Derived class2
    def func3(self):
        print("This function is in child 2.")
```

```
# Driver's code
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

# Encapsulation

Encapsulation in Python describes the concept of bundling data and methods within a single unit. So, for example, when you create a class, it means you are implementing encapsulation.



# Polymorphism

Polymorphism is taken from the Greek words Poly (many) and morphism (forms). It means that the same function name can be used for different types.

```
#Built in Polymorphic function
print(len("Aksadur Rahman"))
print(len([10, 20, 30]))

#User define polymorphic function
def add(x, y, z=0):
    return x+y+z

print(add(30, 20))
print(add(10, 30, 20))
```