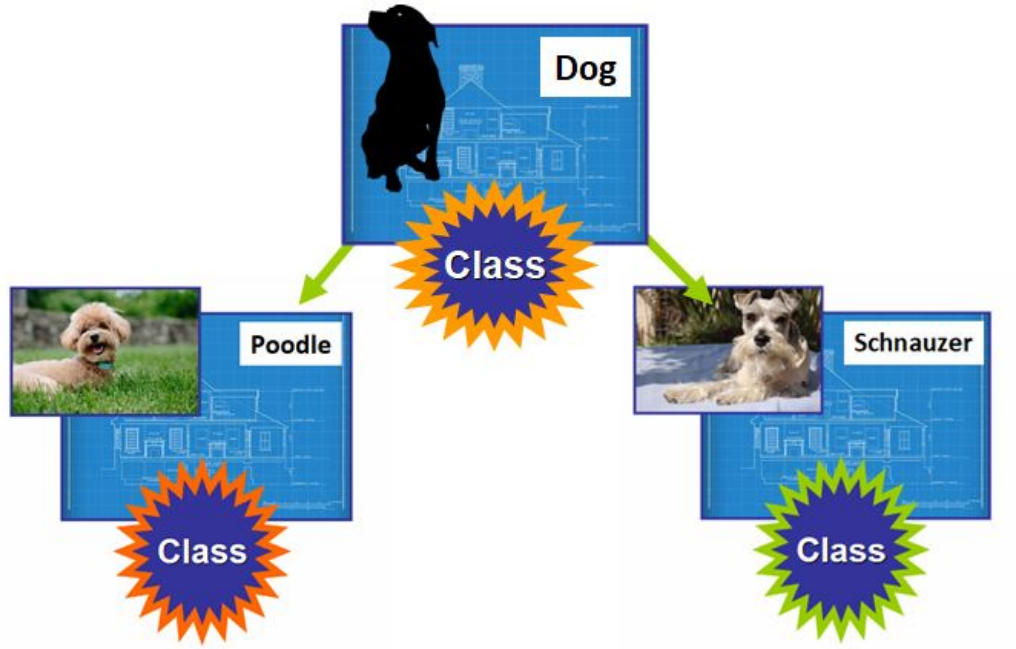
INHERITENCE

Parent Class (Superclass)
The class from which other classes inherit attributes and methods (e.g. Dog).

Child Class (Subclass)
The class that inherits attributes and methods from another class (e.g. Poodle and Schnauzer).

🚩 **Note:** The child class (subclass) inherits from the parent class (superclass).

💡 **Important:** In Python, every class is derived from the object class.



[Basic syntax]

```
class Superclass:
        # Body

class Subclass(Superclass):
        # body
```

example:

```
class Dog:

    def __init__(self, name, age, color):
        self.name = name
        self.age = age
        self.color = color

class Poodle(Dog):
```

```
    def poodle_introduction(self):
        print(f"Hi my name is {self.name} and my age is {self.age}")

poodle1 = Poodle("Snoopy",24,"blue")
poodle1.poodle_introduction()
```

OUTPUT

```
Hi my name is Snoopy and my age is 24
```

issubclass

You can check if <class1> is a subclass of <class2> using this function, as you can see below:

```
print(issubclass(Poodle, Dog))
print(issubclass(Poodle, Poodle))
```

OUTPUT

```
True
True
```

How to inherit using the the __init__()

```
class Vehicle:

    def __init__(self, speed, miles, price):
        self.speed = speed
        self.miles = miles
        self.price = price

class Truck(Vehicle):

    def __init__(self, speed, miles, price, driver):
        super().__init__(speed, miles, price)
        #Vehicle.__init__(self, speed, miles, price)

        self.driver = driver
```

another way was *Vehicle.__init__(self, speed, miles, price)*

How to use "super" to refer to the superclass

super()

This is a built-in Python function that is used to **refer to the immediate parent class** of the current class.

"In a class hierarchy with single inheritance, *super* can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of *super* in other programming languages." — source: Built-in Functions - Python Documentation

Alternative Syntax

You can use super() in __init__() to make your subclass inherit the attributes of the superclass.

For example:

```
>>> class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
>>> class Poodle(Dog):
    def __init__(self, name, age, code):
        super().__init__(name, age)
        self.code = code
```

This line:

```
super().__init__(name, age)
```

is equivalent to the syntax

```
Dog.__init__(self, name, age)
```

💡 **Note:** for this new syntax, **you do not need to pass self** as the first argument. You only pass the other arguments.

Example

```
class Vehicle:

    def __init__(self, speed, miles, price):
        self.speed = speed
        self.miles = miles
        self.price = price


    def hello(self):
        print(f"hello at {self.speed}")
        print(f"hello at {self.speed} and {self.miles}")

class Truck(Vehicle):

    def __init__(self, speed):
        #Vehicle.__init__(self, speed, miles, price)

        self.speed = speed

t1 = Truck(24)
print(t1.speed)
t1.hello()
```

OUTPUT

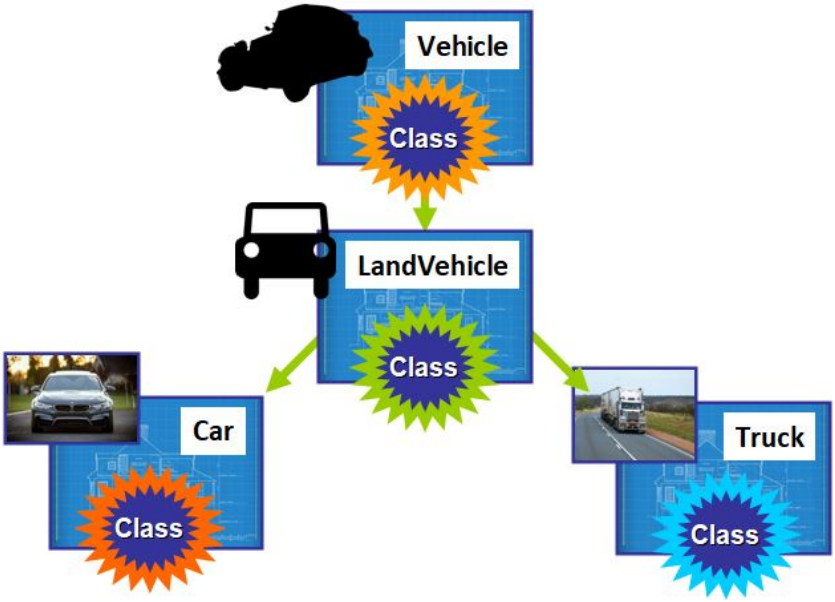
```
Hi my name is Snoopy and my age is 24
24
hello at 24
Traceback (most recent call last):
  File "inherit.py", line 37, in <module>
    t1.hello()
  File "inherit.py", line 26, in hello
    print(f"hello at {self.speed} and {self.miles}")
AttributeError: 'Truck' object has no attribute 'miles'
```

Hence in this case without the super().__init__ ,Truck has only an attribute of speed

Multilevel Inheritance

Multilevel Inheritance

With the syntax that you just learned, you can create more complex hierarchies with multiple levels.



Here we have this hierarchy represented in code:

```
>>> class Vehicle:
        pass
>>> class LandVehicle(Vehicle):
        pass
>>> class Car(LandVehicle):
        pass
>>> class Truck(LandVehicle):
        pass
```

💡 **Note:** For illustration purposes, the classes have an empty body with only a pass statement as a placeholder for the body of the class.

🚩 **Important:** Multilevel inheritance is different from multiple inheritance, in which a class is derived from more than one subclass.

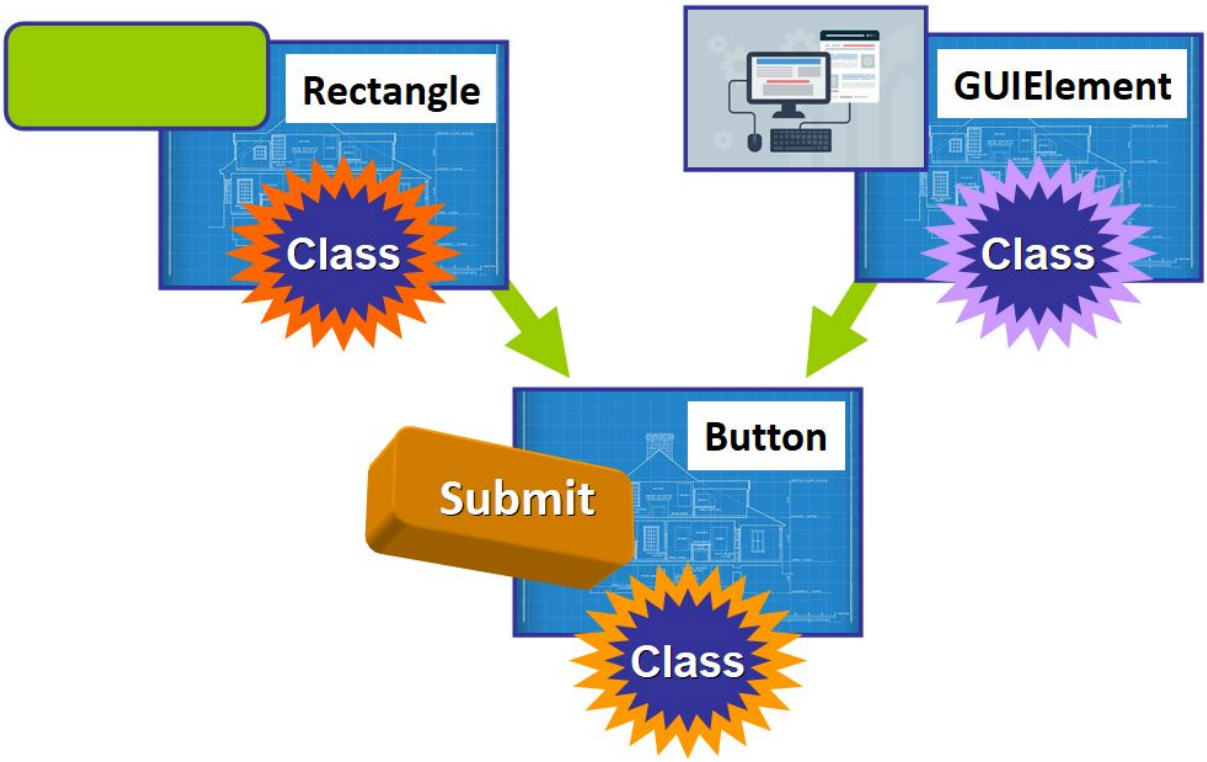
Multiple Inheritance

Multiple Inheritance

Now we will briefly touch the topic of multiple inheritance.

In multiple inheritance, **a class has more than one parent class**.

For example, if you are developing a GUI (Graphical User Interface), a Button class could inherit from **both** the Rectangle class (for style) and from the GUIEelement class (for functionality).

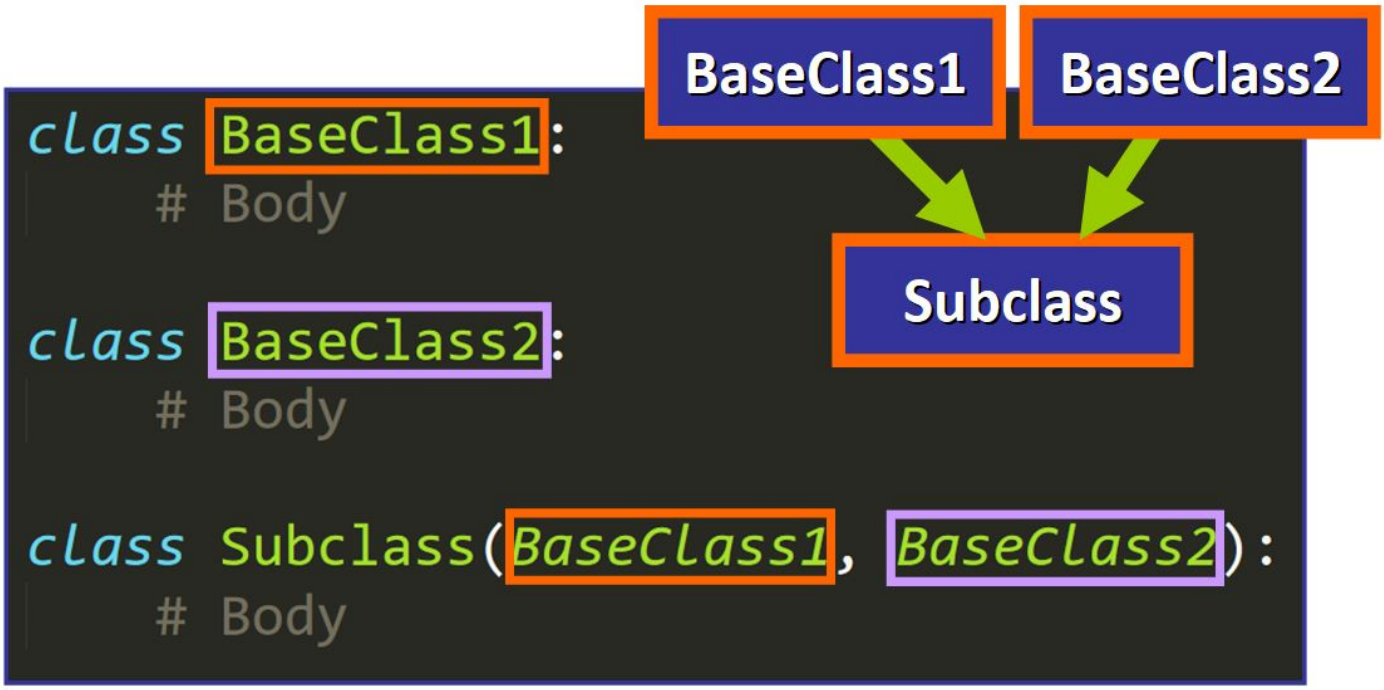


This a very simple example in Python code:

```
class Rectangle:
    def __init__(self, length, width, color):
        self.length = length
        self.width = width
        self.color = color
class GUIElement:
    def click(self):
        print("The object was clicked...")
class Button(Rectangle, GUIElement):
    def __init__(self, length, width, color, text):
        Rectangle.__init__(self, length, width, color)
        self.text = text
```

Syntax

This is the basic syntax to set up multiple inheritance. The subclass will inherit the attributes and methods from both superclasses (base classes).



Method Inheritence

How to Call a Method of the Superclass

You can call a method from the superclass in the subclass with this syntax:

<ClassName>.<method_name>(self, <args>) **e.g** Triangle.find_area(self)

or

super().<method_name>(<args>) **e.g** super().find_area()

Here is an example:

```
>>> class Triangle:
        def __init__(self, base, height):
                self.base = base
                self.height = height
        def find_area(self):
                print((self.base * self.height)/2)
>>> class RightTriangle(Triangle):
        def display_area(self):
                print("=== Right Triangle Area ===")
                # This line calls the method from the Triangle class
                Triangle.find_area(self) # You could also use super().find_area()
>>> right_triangle = RightTriangle(5, 6)
>>> right_triangle.display_area()
=== Right Triangle Area ===
15.0
```

example:

```
class Teacher:

    def __init__(self, name, age, class_code):
        self.name = name
        self.age = age
        self.class_code = class_code

    def welcome_students(self):
        print("Welcome, dear students...")

class PhysicsTeacher(Teacher):

    def welcome(self):
        super().welcome_students()
        #Teacher.welcome_students(self)
        print("Let's start our physics class")

class BiologyTeacher(Teacher):

    def welcome(self):
        super().welcome_students()
        print("Let's start our biology class")

b1 = BiologyTeacher("aksh", 12, 404)
b1.welcome()
```

OUTPUT

```
Welcome, dear students...
Let's start our biology class
```

Method Overriding

If both the subclass and the superclass have the same method name then the subclass method name will be called.we can call the super class method using the **super** keyword

syntax

```
super().method(args) or
superclass.method(self, args)
```

example:

```
class Teacher:

    def __init__(self, name, age, class_code):
        self.name = name
        self.age = age
        self.class_code = class_code

    def welcome_students(self):
        print("Welcome, dear students...")
```

```
class PhysicsTeacher(Teacher):

    def welcome_students(self):
        super().welcome_students()
        print("Let's start our physics class")

class BiologyTeacher(Teacher):

    def welcome_students(self):
        super().welcome_students()
        print("Let's start our biology class")

b1 = BiologyTeacher("aksh", 12, 404)
b1.welcome_students()
```

OUTPUT

```
Welcome, dear students...
Let's start our biology class
```

Concept

Method Overloading is a very common term that you will find and use when you work with Object-Oriented Programming.

It basically occurs when two methods in the same class have the same name but different parameters, so when you call the method, the version that is executed is determined by the data types of the arguments or by the number of arguments.

In Python

🚩 **Python does not support method overloading**. The closest thing that you could think of in Python is default arguments, because you can call a method passing a different number of arguments if you want to use the default values. But this is not method overloading per se.

In Java

Java does support method overloading because you need to explicitly declare the data type of each argument, so the compiler can match the number, sequence, and data types of the arguments with the number, sequence, and data types of the formal parameters to determine which version of the method it should call.

Example in Java:

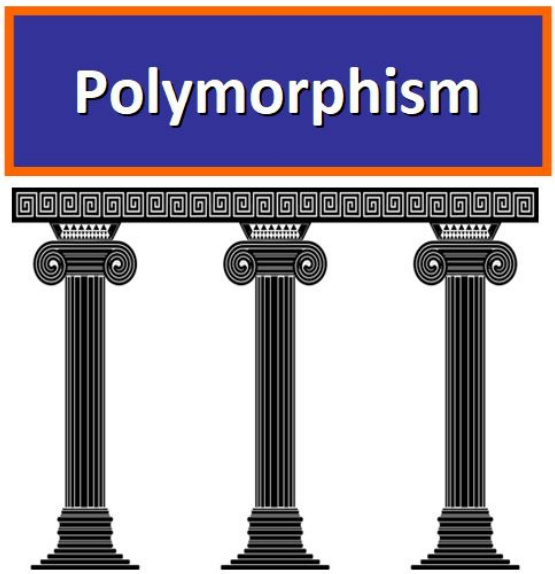
```
class Test {
   public int add(int a, int b) {
       return a + b;
   }
   public int add(int a, int b, int c) {
       return a + b + c;
```

```
    }
}
class Main {
   public static void main(String args[]) {
       Test obj = new Test();
       obj.add(10, 10);  # This will call the first method. Only two arguments
       obj.add(20, 12, 5); # This will call the second method. Three arguments.
   }
}
```

Polymorphism

This is another fundamental pillar of Object-Oriented Programming:

Polymorphism means that **an object can take many forms.**
💡 **Tip:** Polymorphism can be achieved through **method overriding and method overloading** (method overloading is not supported in Python per se).



🔸 Inheritance

In Python, polymorphism can be implemented through inheritance when you override methods from the superclass.

This is an example. First we have the classes File, PDFFile, and TextFile (they both inherit from File).

```
# Classes
class File:
    def __init__(self, name, extension):
        self.name = name
        self.extension = extension
    def open():
        print("Opening a generic file...")
class PDFFile(File):
    def __init__(self, name):
```

```
        File.__init__(self, name, ".pdf")
    def open(self):
        print("Opening a PDF File...")
class TextFile(File):
    def __init__(self, name):
        File.__init__(self, name, ".txt")
    def open(self):
        print("Opening a Text File...")
```

Then we have this function:

```
def open_files(files):
    for file in files:
        file.open()
```

If we create instances of these classes and include them in a list:

```
pdf1 = PDFFile("Brochure")
pdf2 = PDFFile("Course Advertising")
text1 = TextFile("List of Students")
files = [pdf1, text1, pdf2]
```

We can call the function open_files passing this list as argument:

```
open_files(files)
```

⚠️ **Important:** don't you notice something very peculiar? We are passing a list that contains objects of different types and the code works correctly for all of them. This is because we know that all the classes