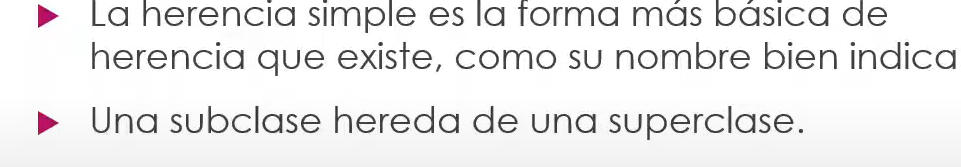
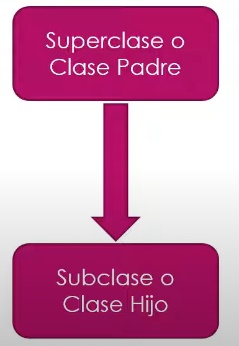
# **Pilares de la programación orientada a objetos [ejercicios]**

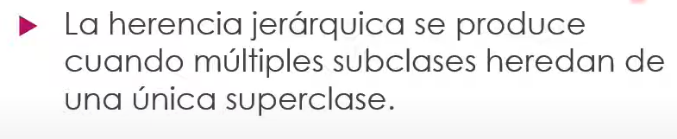
## **Herencia:**

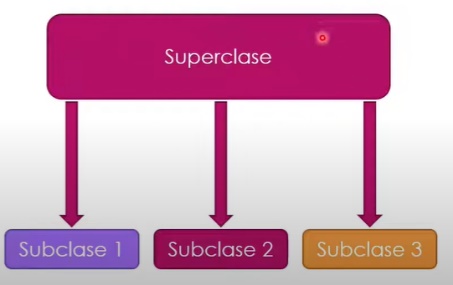
* 1. **Herencia simple:**



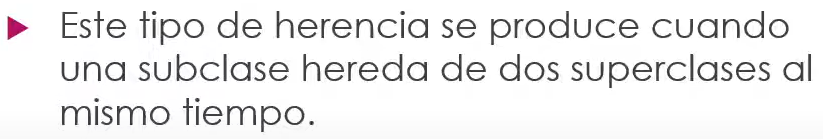


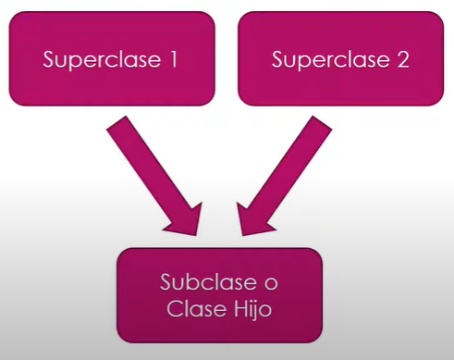
* 1. **Herencia jerárquica**



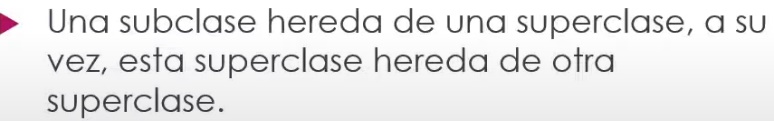


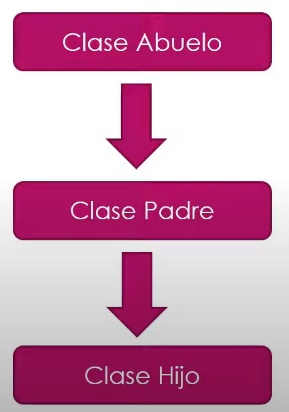
* 1. **Herencia Múltiple**



****

* 1. **Herencia Multinivel**

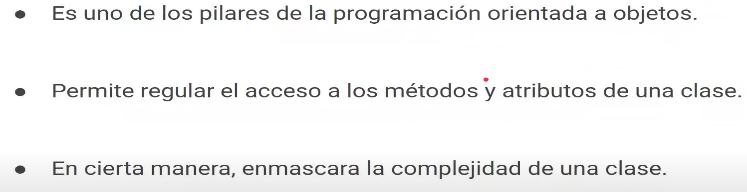
****

****

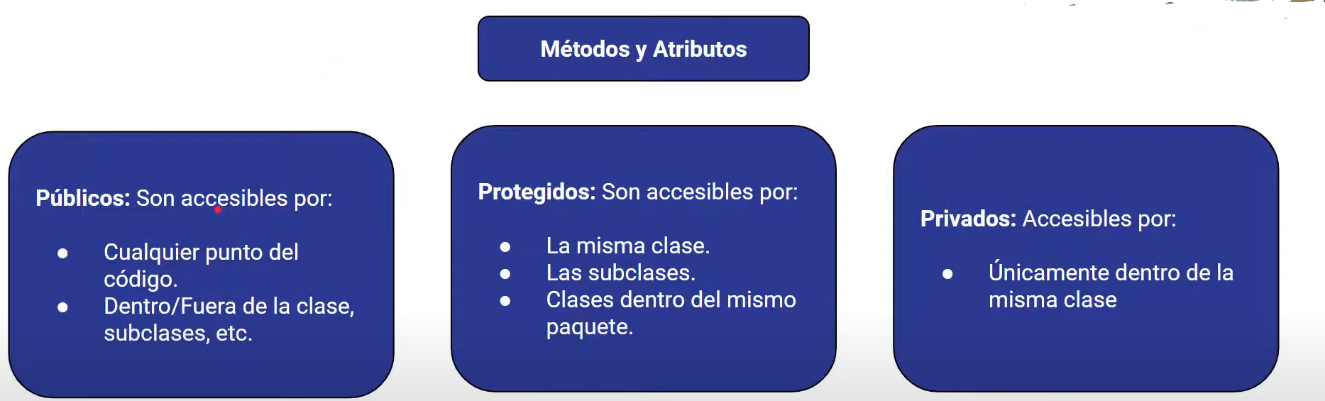
**Ejercicio Practico**

class Persona:  
 def \_\_init\_\_(self,nombre,edad,DNI):  
 self.nombre = nombre  
 self.edad = edad  
 self.dni = DNI  
 def presentarse(self):  
 print(f"Hola! me llamo {self.nombre} y tengo {self.edad} años")  
class Trabajador(Persona):  
 def \_\_init\_\_(self,nombre,edad,DNI, sueldo, cargo, empresa):  
 super().\_\_init\_\_(nombre,edad,DNI)  
 self.sueldo =sueldo  
 self.cargo = cargo  
 self.empresa = empresa  
 def calcularSueldoAnual(self):  
 return 12\*self.sueldo + 2000  
  
class Estudiante(Persona):  
 def \_\_init\_\_(self,nombre,edad, DNI,universidad, curso,asignatura):  
 super().\_\_init\_\_(nombre,edad, DNI)  
 self.universidad = universidad  
 self.curso =curso  
 self.asignatura = asignatura  
 def describirse(self):  
 print(f"Hola soy {self.nombre}!. Tengo {self.edad} y estudio en la universidad {self.universidad}")  
  
trabajador1 = Trabajador("Juan",24,"eqwewq",1500,"machaca","google")  
trabajador1.presentarse()  
print(trabajador1.calcularSueldoAnual())  
  
estudiante1 = Estudiante("Maria",20,"12312321k","Universidad de madrid",3, ["programacion","calculo","algebra"])  
estudiante1.describirse()

## **Encapsulación:**

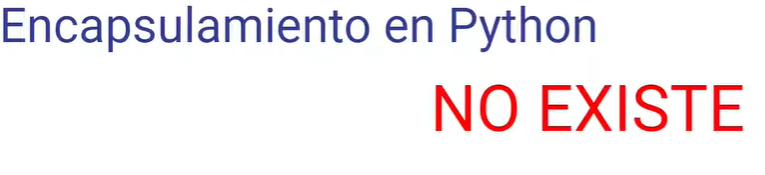


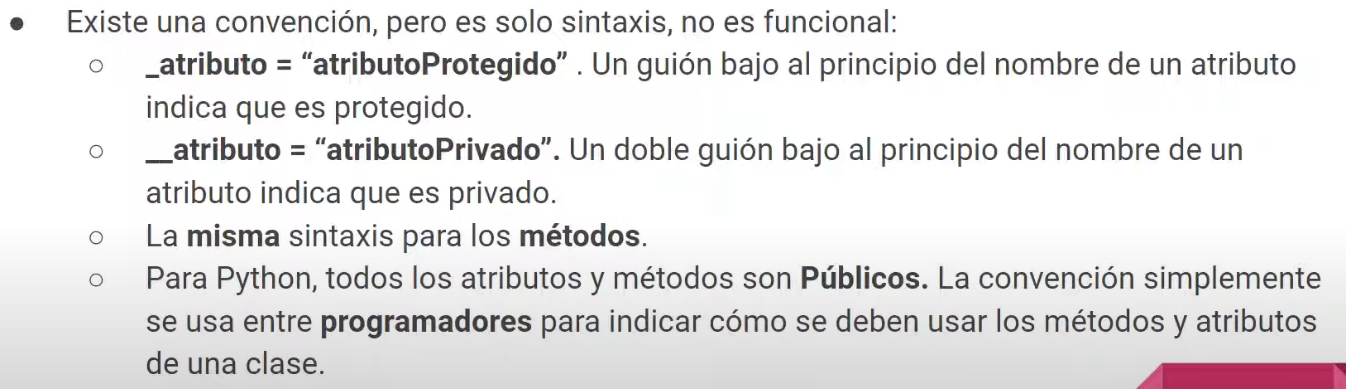
**Encapsulamiento: Modificadores de Acceso**

****

**Utilidades del encapsulamiento:**

* Nos permite ocultar métodos y atributos fuera de la propia clase
* Podemos regular la modificación de los atributos (privados) evitando que se accedan a ellos directamente. Crearemos métodos (públicos) para modificar los atributos de un objeto.
* Enmascarar la complejidad de algunos métodos haciéndolos privados y utilizarlos desde métodos públicos.

****

****

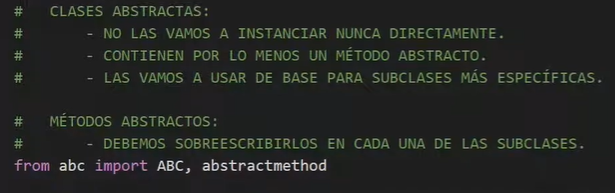
**Ejemplo 1:**

class Circulo:  
 def \_\_init\_\_(self,radio):  
 self.\_\_radio = radio  
 self.\_\_pi = 3.1415  
 def calcularPerimetro(self):  
 return 2 \* self.\_\_pi \* self.\_\_radio  
 def calcularArea(self):  
 return self.\_\_pi \* self.\_\_radio\*\*2  
 def getPi(self):  
 return self.\_\_pi  
 def setRadio(self,nuevo):  
 if type(nuevo) == int or type(nuevo) == float:  
 if nuevo>0:  
 self.\_\_radio = nuevo  
 print(f"El radio se ha modificado correctamente: {self.\_\_radio}")  
 else:  
 print("El radio no puede ser negativo")  
 else:  
 print("El radio tiene que ser un numero positivo")  
c1 = Circulo(2.5)  
print(c1.calcularArea())  
print(c1.calcularPerimetro())  
print(f"La constante PI es {c1.getPi()}")  
c1.setRadio(34)  
c1.setRadio("hola")  
c1.setRadio(-1)  
  
#atributo: \_\_radio -> \_circulo\_\_radio . En general \_NombreDeLaClase\_\_nombreDelAtributo

## **Polimorfismo:**

class Empleado:  
 def \_\_init\_\_(self,nombre,sueldoMensual):  
 self.nombre = nombre  
 self.sueldoMensual = sueldoMensual  
 def calcularSueldoAnual(self):  
 sueldo = 12\*self.sueldoMensual \* (1 \* 1/100)  
 print(f"El sueldo anual de {self.nombre}, empleado normal, es de {sueldo} $")  
class Contable(Empleado):  
 def calcularSueldoAnual(self):  
 sueldo = 12\*self.sueldoMensual \* (1 \* 4/100)  
 print(f"El sueldo anual de {self.nombre}, Contable, es de {sueldo} $")  
class Publicista(Empleado):  
 def calcularSueldoAnual(self):  
 sueldo = 12\*self.sueldoMensual \* (1 \* 5/100)  
 print(f"El sueldo anual de {self.nombre}, Publicista, es de {sueldo} $")  
class Becario(Empleado):  
 def calcularSueldoAnual(self):  
 sueldo = 12\*self.sueldoMensual  
 print(f"El sueldo anual de {self.nombre}, Publicista, es de {sueldo} $")  
  
empleados = [  
 Empleado("Juan",1000),  
 Contable("Angela",1100),  
 Publicista("Ryan",1200),  
 Becario("Pepito",750)  
]  
for empleado in empleados:  
 empleado.calcularSueldoAnual()

# **Abstracción**



from abc import ABC, abstractmethod  
  
class Personaje(ABC):  
 @abstractmethod  
 def \_\_init\_\_(self,nombre):  
 self.nombre = nombre  
 self.nivel = 0  
 self.inventario = []  
 self.vida = 100  
 @abstractmethod  
 def atacar(self,objetivo):  
 pass  
 @abstractmethod  
 def getestatus(self):  
 print(f"Nombre: {self.nombre}")  
 print(f"Nivel: {self.nivel}")  
  
 def subirDeNivel(self):  
 self.nivel+=1  
  
 def verInventario(self):  
 print(f"Inventario de {self.nombre}")  
 for objeto in self.inventario:  
 print(objeto)  
class Mago(Personaje):  
 def \_\_init\_\_(self,nombre):  
 super().\_\_init\_\_(nombre)  
 self.vida = 100  
 self.inteligencia = 95  
 self.inventario = ["grimorio","Pocion de mana"]  
  
 def getestatus(self):  
 print("Es de la clase Mago")  
 super().getestatus()  
 def atacar(self,objetivo):  
 objetivo.vida -=self.inteligencia\*0.6  
 print(f"Vida actual del objetivo enemigo {objetivo.nombre} es {objetivo.vida}")  
class Guerrero(Personaje):  
 def \_\_init\_\_(self,nombre):  
 super().\_\_init\_\_(nombre)  
 self.vida = 200  
 self.fuerza = 75  
 self.inventario = ["Pocion de vida","escudo","espada"]  
 def getestatus(self):  
 print("Es la clase guerrero")  
 super().getestatus()  
 def atacar(self,objetivo):  
 objetivo.vida -= self.fuerza \* 0.8  
 print(f"Vida actual del objetivo enemigo {objetivo.nombre} es {objetivo.vida}")  
  
guerrero = Guerrero("Kaladin")  
mago = Mago("Yuno")  
print()  
guerrero.getestatus()  
mago.getestatus()  
print()  
guerrero.verInventario()  
mago.verInventario()  
print()  
mago.atacar(guerrero)  
guerrero.atacar(mago)

# **Ejercicio final**