Relaciones en la POO

En el mundo de la Programación Orientada a Objetos tenemos diferentes tipos de relaciones en las que cada una tiene un significado semántico y una consecuencia en código. Veremos cómo implementar cada una de ellas en Python.

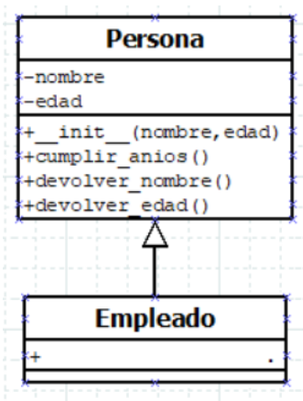
Herencia

Le permite a una clase (subclase) heredar las características y el comportamiento de otra clase (superclase).

Sintaxis:

```
class Subclase (Superclase):
```

Ejemplo:




```
class Persona:   #esta es la superclase

  def __init__(self, nombre, edad):  #constructor de la superclase
    self.__nombre = nombre
    self.__edad = edad

  def cumplir_anios(self):
    self.__edad += 1

  def devolver_nombre(self):
    return self.__nombre

  def devolver_edad(self):
    return self.__edad
########################################################################
class Empleado(Persona):  #esta es la subclase Empleado que hereda de Persona
  pass

########################################
emp = Empleado ("Juan", 30)  #se crea un empleado (sublcase)
emp.cumplir_anios() #se llama al método heredado de la superclase

emp.salario = 1_500_000
print (f"nombre del empleado: {emp.devolver_nombre()}")
print (f"edad del empleado: {emp.devolver_edad()}")
print (f"salario del empleado: {emp.salario}")
```

```
nombre del empleado: Juan
edad del empleado: 31
salario del empleado: 1500000
```

super()

La función super() se utiliza comúnmente para acceder a métodos y atributos de una clase padre desde una clase hija.

```
class Persona:

  def __init__(self, nombre, edad):  #constructor de la superclase
    self.__nombre = nombre
    self.__edad = edad

  def cumplir_anios(self):
    self.__edad += 1
```

```
  def devolver_nombre(self):
    return self.__nombre

  def devolver_edad(self):
    return self.__edad

class Empleado(Persona):
  def __init__(self, nombre, edad, salario): #constructor de la subclase
    super().__init__(nombre, edad)  #envía el nombre y edad al constructor de la superclase
    self.__salario = salario

  def devolver_salario(self):
    return self.__salario

########################################
emp = Empleado ("Juan", 30, 1_500_000)  #se crea un empleado (sublcase)
emp.cumplir_anios()
print (f"nombre del empleado: {emp.devolver_nombre()}")
print (f"edad del empleado: {emp.devolver_edad()}")
print (f"salario del empleado: {emp.devolver_salario()}")
```

```
nombre del empleado: Juan
edad del empleado: 31
salario del empleado: 1500000
```

issubclass()

issubclass() es una función que nos permite comprobar si una clase es subclase de otra.

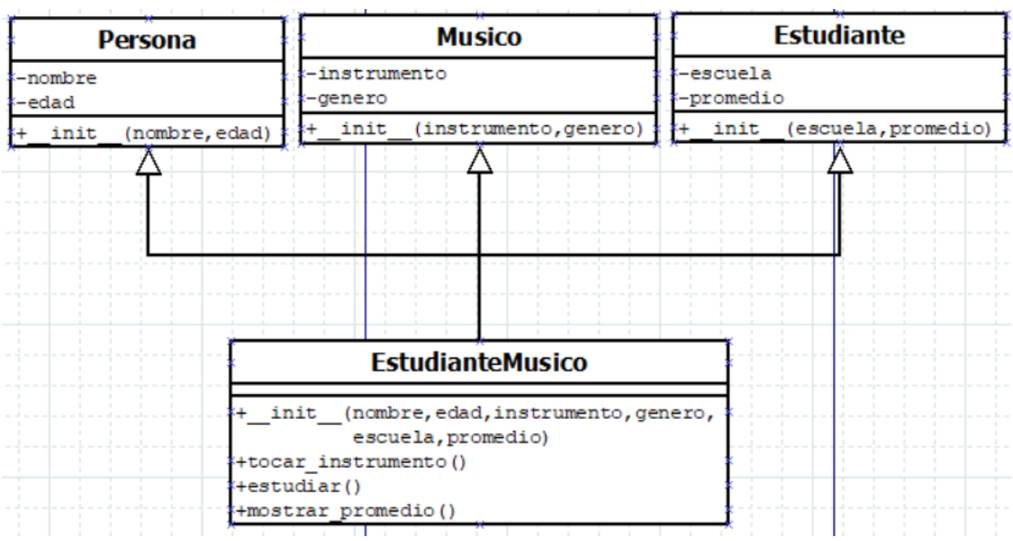
```
print (issubclass(Empleado, Persona))
print (issubclass(Persona, Empleado))
```

```
True
False
```

Herencia múltiple

En Python la herencia múltiple es permitida y se define simplemente enviando varias superclases entre paréntesis al momento de crear una subclase

```
class Subclase (Superclase1, Superclase2, Superclase_n):
    pass
```



```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

class Musico:
    def __init__(self, instrumento, genero):
        self.instrumento = instrumento
        self.genero = genero

class Estudiante:
    def __init__(self, escuela, promedio):
        self.escuela = escuela
        self.promedio = promedio

class EstudianteMusico(Persona, Estudiante, Musico):
    def __init__(self, nombre, edad, instrumento, genero, escuela, promedio):
        Persona.__init__(self, nombre, edad)
        Estudiante.__init__(self, escuela, promedio)
        Musico.__init__(self, instrumento, genero)

    def tocar_instrumento(self):
        print(f"{self.nombre} está tocando {self.instrumento}.")

    def estudiar(self):
        print(f"{self.nombre} está estudiando en {self.escuela}.")

    def mostrar_promedio(self):
        print(f"{self.nombre} tiene un promedio de {self.promedio}.")
########################################################################
est_musico = EstudianteMusico("Diego",28,"Violín","balada","bellas artes",4.9)
est_musico.tocar_instrumento()
est_musico.estudiar()
est_musico.mostrar_promedio()
```

```
Diego está tocando Violín.
Diego está estudiando en bellas artes.
Diego tiene un promedio de 4.9.
```

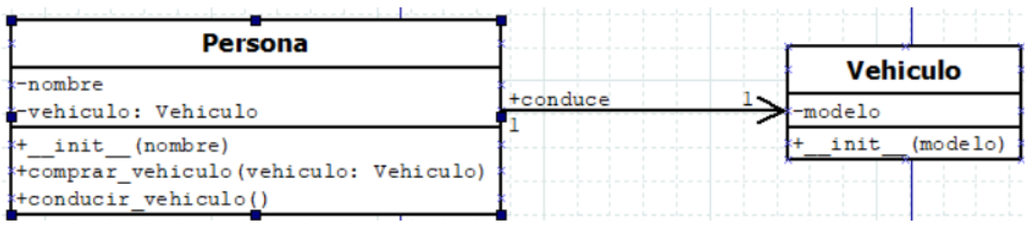
Asociación

Una asociación es una relación semántica entre objetos y se da cuando un objeto conoce a otro objeto y por lo tanto puede acceder a su comportamiento.

Asociaciones Unidireccionales

Si la asociación es unidireccional, solo el objeto A conoce al objeto B, pero si es bidireccional tanto A como B se conocen.

En la siguiente asociación entre Persona y Vehiculo, la Persona conoce el Vehiculo y por lo tanto, puede acceder a su comportamiento, mas el Vehiculo no conoce a la Persona



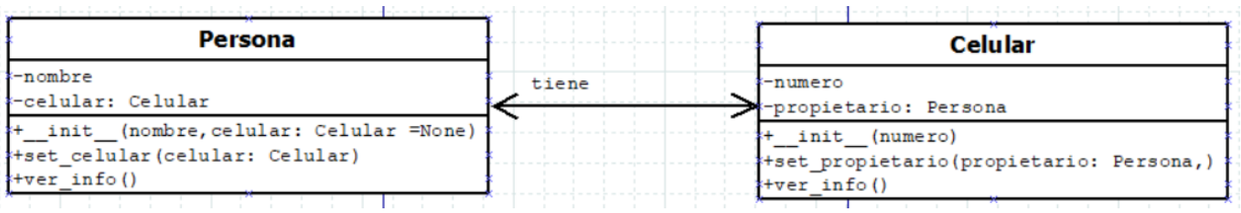

```
class Vehiculo:
    def __init__(self, modelo):
        self.modelo = modelo

class Persona:
    def __init__(self, nombre):
        self.nombre = nombre
        self.vehiculo = None #inicialmente, la persona no tiene un vehículo

    def comprar_vehiculo(self, vehiculo:Vehiculo):
        self.vehiculo = vehiculo #Asocio un objeto vehiculo al atributo vehículo

    def conducir_vehiculo(self):
        if self.vehiculo is not None: #Verifico si la persona tiene un vehículo
            print(f"{self.nombre} está conduciendo un vehículo {self.vehiculo.modelo}")
        else:
            print(f"{self.nombre} no tiene un vehículo para conducir")
########################################################################

vehiculo1 = Vehiculo("Toyota Corolla") #creo una instancia de vehiculo
juan = Persona("Juan")             # creo una instancia de persona

juan.comprar_vehiculo(vehiculo1)   #juan compra el vehiculo

juan.conducir_vehiculo()  #juan intenta conducir el vehiculo

pedro = Persona("Pedro")
pedro.conducir_vehiculo()
```

```
Juan está conduciendo un vehículo Toyota Corolla
Pedro no tiene un vehículo para conducir
```

Asociaciones Bidireccionales

Si la asociación es bidireccional, el objeto A conoce al objeto B y el objeto B conoce al objeto A.

En la siguiente asociación entre Persona y Celular, la Persona conoce el Celular asignado y por lo tanto, puede acceder a su comportamiento, además, el Celular conoce a la Persona (propietario).



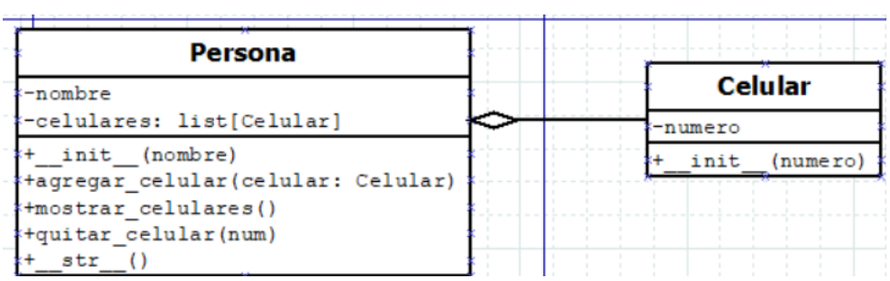

```
class Persona:
    def __init__(self, nombre, celular=None):
        self.nombre = nombre
        if celular is not None:
            self.set_celular(celular) #asociar el celular a la persona

    def set_celular(self, celular):
        self.celular = celular
        celular.set_propietario(self)  # Establece la persona como propietaria del celular

    def ver_info(self):
        if self.celular is not None:
            return f"{self.nombre} tiene el celular {self.celular.numero}"
        else:
            return f"{self.nombre} no tiene celular"


class Celular:
    def __init__(self, numero):
        self.numero = numero
        self.propietario = None  #inicialmente, el celular no tiene propietario

    def set_propietario(self, propietario:Persona):
        self.propietario = propietario # Asocia la persona como propietaria del celular

    def ver_info(self):
        if self.propietario is not None:
            return f"El celular {self.numero} pertenece a {self.propietario.nombre}"
        else:
            return f"El celular {self.numero} no tiene propietario"
######################################################################

c = Celular (3002350020) # Crear una instancia de Celular
juan = Persona ("Juan", c)# Crear una instancia de Persona y asociar el celul
print("Informacion de la persona ",juan.ver_info())
print("Informacion del celular ",c.ver_info())
```

```
Informacion de la persona  Juan tiene el celular 3002350020
Informacion del celular  El celular 3002350020 pertenece a Juan
```

Agregación

Una Agregación es una relación entre dos clases en la que una clase contiene una o varias instancias de otra clase como parte de su estructura. La clase que contiene las instancias de otra clase se llama clase contenedora o clase agregadora, mientras que la clase contenida se llama clase contenida o clase agregada.

En el siguiente ejemplo, una instancia de la clase Persona contiene varios objetos de tipo Celular



```
class Celular:
    def __init__(self, numero):
        self.numero = numero

class Persona:
    def __init__(self, nombre):
        self.nombre = nombre
        self.celulares = [] # Lista para almacenar los celulares asociados a la persona

    def agregar_celular(self, celular:Celular):
        self.celulares.append(celular) # Agrega un celular a la lista de celulares de la persona

    def mostrar_celulares(self):
        print ("celulares de ", self)
        for cel in self.celulares:
          print (cel.numero)

    def quitar_celular(self, num):
      for c in self.celulares:
        if c.numero == num:
          self.celulares.remove(c) # Elimina un celular de la lista de celulares de la persona

    def __str__(self):
      return self.nombre
#######################################################
p = Persona ("Juan")
c1 = Celular (300248)
c2 = Celular (320789)
c3 = Celular (311445)

p.agregar_celular(c1)
p.agregar_celular(c2)
p.agregar_celular(c3)

p.mostrar_celulares()
p.quitar_celular(320789)
p.mostrar_celulares()

print("numero del cel 2: ", c2.numero)
```

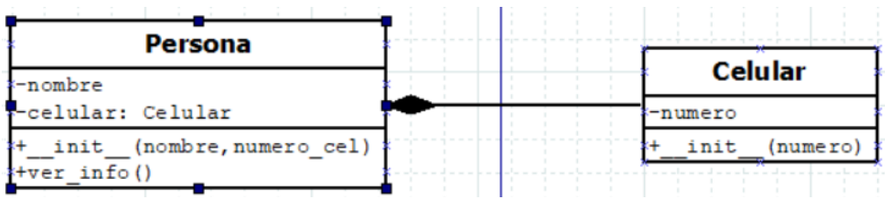
```
celulares de  Juan
300248
320789
311445
celulares de  Juan
300248
311445
numero del cel 2:  320789
```

Nótese que los objetos de tipo Celular existen por fuera de la clase Persona y su existencia no se limita a la existencia de la clase contenedora.

Composición

En POO, la relación de composición es una relación entre dos clases en la que una clase contiene a otra clase como parte de su estructura y es responsable de crear y destruir las instancias de la clase contenida. La clase contenida no puede existir sin la clase contenedora y está completamente encapsulada dentro de ella.

En el siguiente ejemplo, los objetos de la clase Celular son creados en el constructor de la clase Persona, de esta manera no pueden existir de manera independiente y al eliminar una instancia de la clase Persona, se eliminarán todos los objetos Celular contenidos.




```
class Celular:
    def __init__(self, numero):
        self.numero = numero

class Persona:
    def __init__(self, nombre, numero_cel):
        self.nombre = nombre
        self.celular = Celular(numero_cel) #Se crea una instancia de Celular utilizando el número de celular proporcionado

    def ver_info(self):
      print (f"soy {self.nombre} y tengo un celular número {self.celular.numero}")
#################################################################
p = Persona("Juan", 300245) # cuando instancio la clase Persona tambien instancio la clase  celular
p.ver_info()

print (p.celular.numero)
```

```
soy Juan y tengo un celular número 300245
300245
300022
```

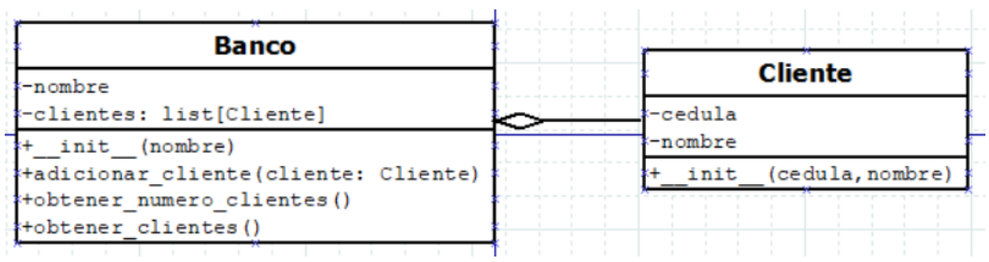
Ahora con varios celulares

```
class Celular:
    def __init__(self, numero):
        self.numero = numero

class Persona:
    lenguaje = "Python" # Atributo de clase


    def __init__(self, nombre):
        self.nombre = nombre
        self.celulares=[]  # Lista para almacenar los celulares asociados a la persona


    def agregar_celular(self, numero):
        self.celulares.append(Celular(numero)) # Agrega una nueva instancia de Celular a la lista

    def mostrar_celulares(self):
        print ("celulares de ", self)
        for cel in self.celulares:
          print (cel.numero)

    def __str__(self): #Método para obtener una representación de cadena de la persona.
      return self.nombre


p = Persona ("Pedro")    #creamos una instancia de persona
p1 = Persona ("Amparo")  #creamos segunda instancia de persona
print (p.nombre)
print (p1.nombre)

print (p.lenguaje )  #mostramos el atributo de clase de las instancias persona
print (p1.lenguaje)

Persona.lenguaje = "Java" #modificamos el atributo de clase


print (p.lenguaje)   #mostramos el cambio del atributo de clase de las instancias persona
print (p1.lenguaje)

print (vars(p)) #mostramos atributos de las instancias
print (vars(p1)) #


p.agregar_celular(320865) # Agregar celulares a la lista de celulares de la primera persona
p.agregar_celular(300546)
p.agregar_celular(310892)

p.mostrar_celulares()   #Mostrar los números de los celulares de la primera persona
print ("segundo celular ", p.celulares[1].numero)
```

```
Pedro
Amparo
Python
Python
Java
Java
{'nombre': 'Pedro', 'celulares': []}
{'nombre': 'Amparo', 'celulares': []}
celulares de  Pedro
320865
300546
310892
segundo celular  300546
```

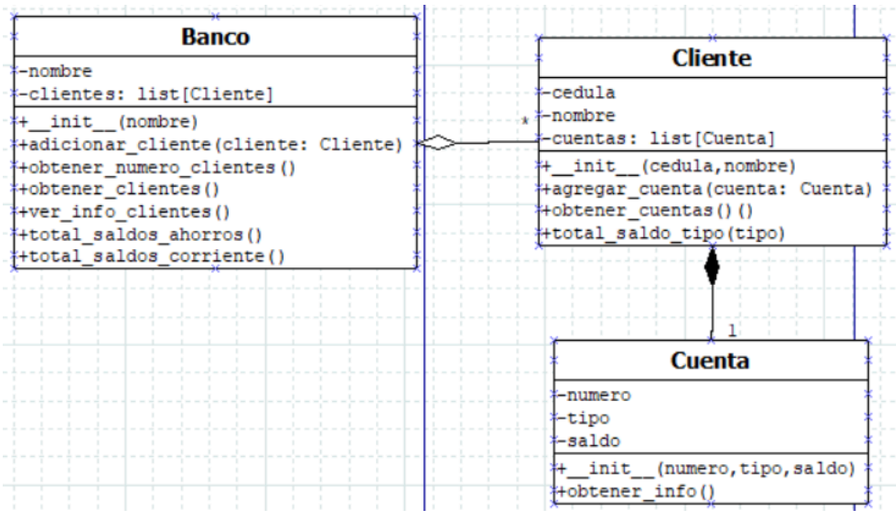
Apropiación

1. A partir del siguiente diagrama:



Cree las clases Banco y Cliente para que respondan al diagrama y compruebe cada uno de sus métodos

2. A partir del siguiente diagrama:



Realice los ajustes necesarios para que:

- En la aplicación solo se pueda tener un único Banco, aunque se le puede cambiar su nombre en cualquier momento
- Se pueda ver la información de todos los clientes del banco incluyendo su cédula, nombre, número de cuenta, tipo y saldo
- Se pueda ver el total de los saldos en las cuentas de ahorros
- Se pueda ver el total de los saldos en las cuentas corrientes

3. Implemente dos relaciones de herencia al ejercicio anterior creando una clase CuentaAhorro y otra CuentaCorriente que hereden de la clase Cuenta. Elimine el atributo tipo de la clase Cuenta y agregue a la clase CuentaAhorro un atributo llamado interés con un método aplicar_interes que incrementa el saldo en el interés dado y a la CuentaCorriente incluir un atributo llamado descuento con un método aplicar_descuento que disminuye el saldo el descuento dado. La aplicación debe continuar funcionando con los ajustes dados.