Unidad 3
Herencia y Polimorfismo

Prof: Lic. Rosales Pablo (prosales@unpata.edu.ar)
Materia: Programación Orientada a Objetos
Año: 2024

Reutilización y Extensibilidad

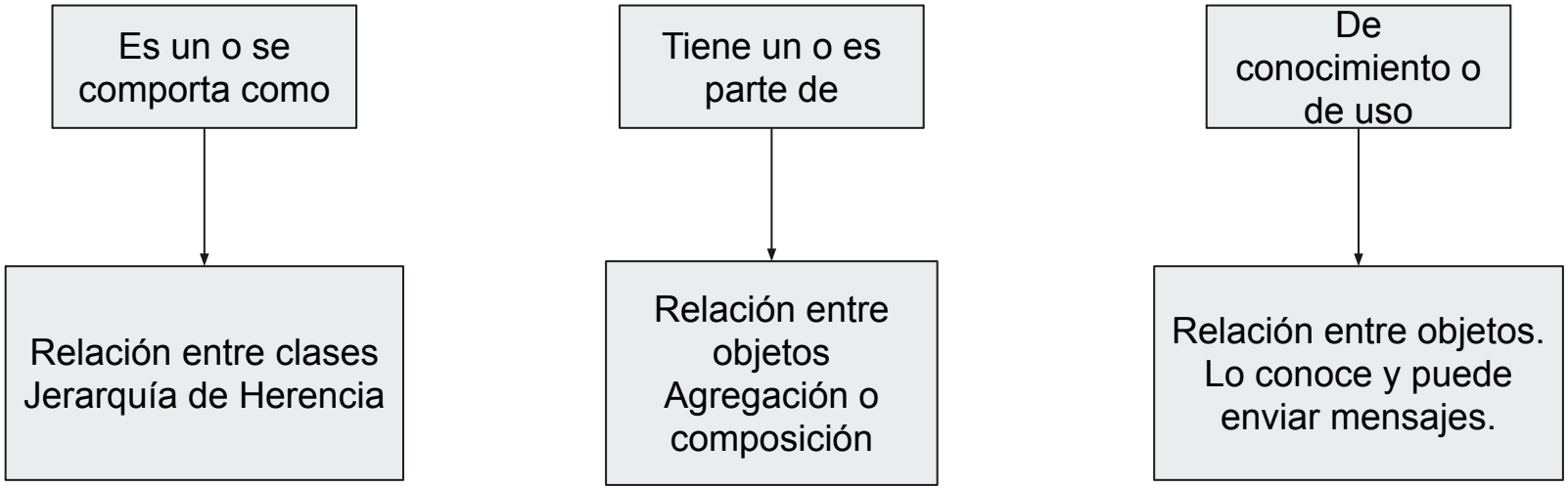
En POO se pueden definir clases que representan objetos que comparten son lo una parte de su estructura y comportamiento, son similares pero no iguales. Esas similitudes se pueden expresar mediante el uso de:

HERENCIA Y POLIMORFISMO

Esto permite reutilizar código, existen funcionalidades que ya fueron implementadas en otros sistemas y que no son necesarias de implementar nuevamente. (ABM, conexiones a BD, Informes, facturaciones etc.)

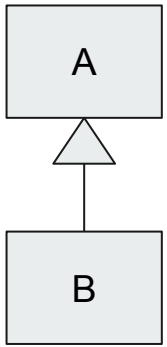
La extensibilidad permite extender del código sin afectarlo.

Relaciones


Es un o se comporta como
Relación entre clases Jerarquía de Herencia
Tiene un o es parte de
Relación entre objetos Agregación o composición
De conocimiento o de uso
Relación entre objetos. Lo conoce y puede enviar mensajes.

Herencia

Superclase / clase base / clase padre



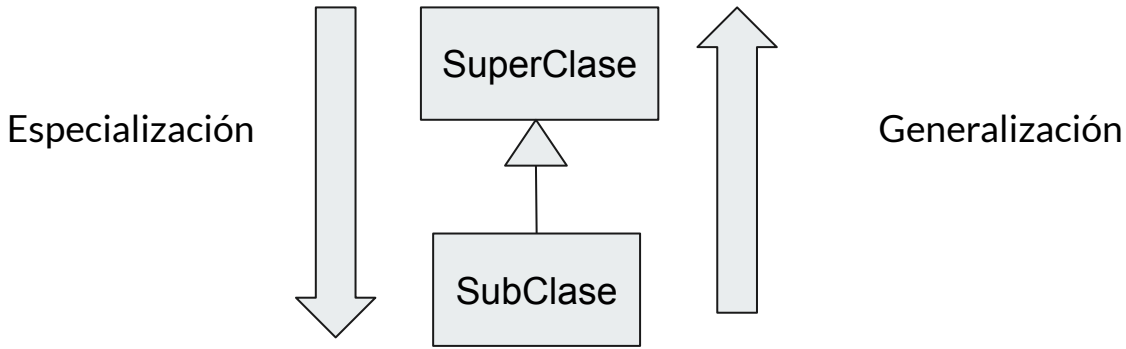
es un ...

Subclase / clase derivada / clase hija

Nos permite derivar una o varias subclases de otra más genérica (superclase)

- La superclase reúne atributos y métodos comunes a las subclases.
- Las subclases definen atributos y métodos propios más específicos.
- la subclase extiende el comportamiento de la superclase de todos los miembros no privados

Herencia Generalización y Especialización




```
class SubClase(SuperClase):
    pass
```

Un hijo puede sustituir al padre en todo momento.

Principio de sustitución de Liskov. La “L” en los principios de diseños orientados a objetos (SOLID)

¿Qué se puede hacer en la subclase?

- **heredar** atributos, métodos y constructor de la superclase
- agregar nuevos atributos y métodos
- **redefinir** (sobrescribir, override) métodos. Dar una nueva implementación.
- **extender** métodos, llamando al método del padre con super() y añadir nuevo comportamiento.
- acceder a métodos del padre con super ()
- llamar al constructor con super().__init__(...)
- **polimorfismo** (a ver en las próximas clases)

fuente: https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html

Constructores y herencia

¿que saldría por consola?

```
class A:
    def __init__(self):
        print("constructor de A")

class B(A):
    def __init__(self):
        super().__init__()
        print("constructor de B")

class C(B):
    def __init__(self):
        super().__init__()
        print("constructor de C")

letra = C()
```

Visibilidad

¿Cúal es la salida por consola?

¿Cómo sería para modificar la visibilidad de los métodos?

```python
class A:
    def __init__(self):
        print("constructor de A")
        self.atributo = "atributo publico de A"
        self._atributo_protected = "atributo protected"
        self.__atributo_private = "atributo private"

class B(A):
    def __init__(self):
        super().__init__()
        print("constructor de B")

mi_b = B()
print(mi_b.atributo)
print(mi_b._atributo_protected)
print(mi_b.__atributo_private)
```

Clases Abstractas

- No se pueden crear instancias de estas clases.
- Contiene métodos que son abstractos, que no están implementados.
- Como cualquier otra superclase, definen características comunes a los objetos que son instancias de la subclases.
- Aumenta la compresión del problema, fomentando la reutilización del código y la extensibilidad
- Fomenta el polimorfismo

```python
from abc import ABC, abstractmethod

class Forma(ABC):
    def __init__(self, color):
        self.color = color

    @abstractmethod
    def area(self):
        pass

class Circulo(Forma):
    def __init__(self, radio, color):
        super().__init__(color)
        self.radio = radio

    def area(self):
        return 3.14 * self.radio * self.radio

#forma = Forma("azul")
circulo = Circulo(5, "verde")
print(circulo.area())
```

¿Qué pasa si no se implementa el método área?

```python
from abc import ABC, abstractmethod

class Forma(ABC):
    def __init__(self, color):
        self.color = color

    @abstractmethod
    def area(self):
        pass

class Circulo(Forma):
    def __init__(self, radio, color):
        super().__init__(color)
        self.radio = radio

    def area(self):
        return 3.14 * self.radio * self.radio

class Rectangulo(Forma):
    def __init__(self, alto, ancho, color):
        super().__init__(color)
        self.alto = alto
        self.ancho = ancho


#forma = Forma("azul")
circulo = Circulo(5, "verde")
print(circulo.area())

rectangulo = Rectangulo(2,3,"azul")
print(rectangulo.area)
```

Sobreescritura / Override

cuando se llama a emitir_sonido(), python busca la implementación en la subclase, en caso de no encontrarla reutiliza la del padre.

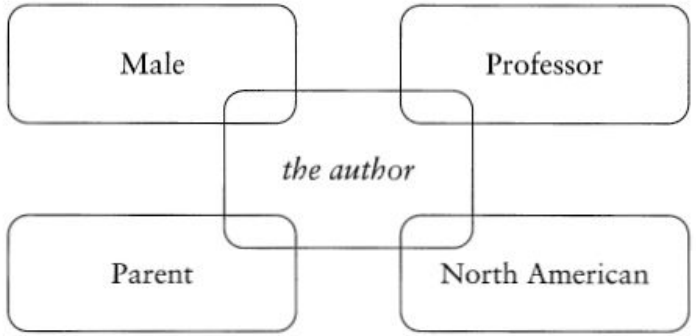
la sobreescritura permite personalizar o hacer más específica la funcionalidad heredada, manteniendo una interfaz consistente en la estructura de herencia.

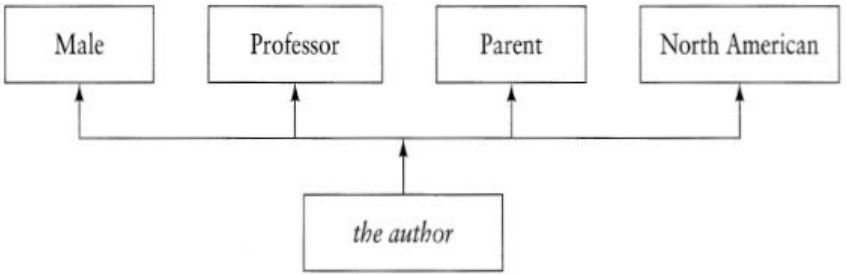
Herencia múltiple

“Como regla, si pensas que necesitas herencia múltiple, probablemente estés equivocado, pero si sabes que lo necesitas, probablemente estés en lo cierto.”

“A puede ser visto como un B”.

el autor puede ser visto como un hombre, profesor, padre, norteamericano...





Fuente: An introduction to Object-Oriented Programming. Timothy A. Budd

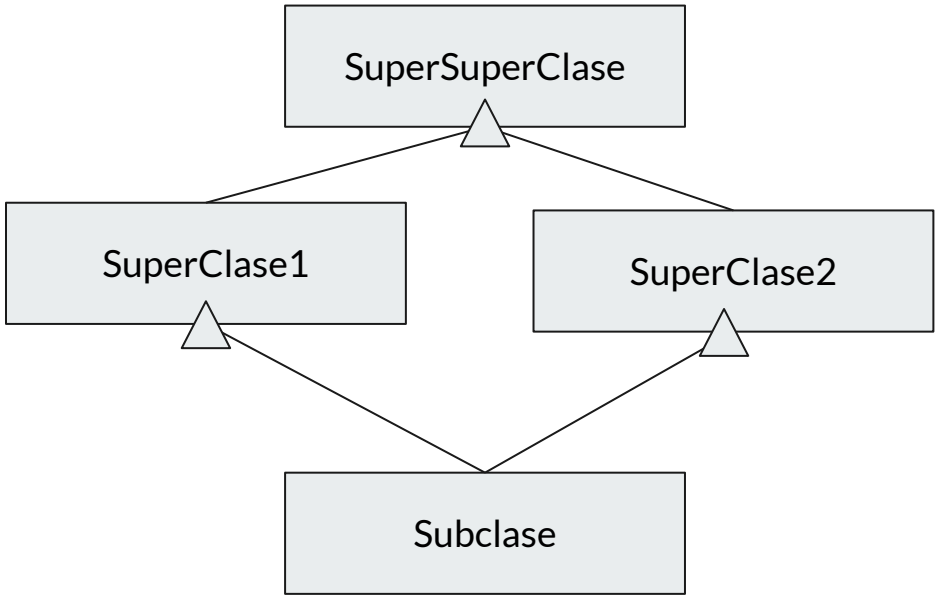
Herencia múltiple

```python
class Padre():
  def __init__(self):
    print("hola desde clase padre")

class Madre():
  def __init__(self):
    print("hola desde la clase madre")

class Hijo(Padre,Madre):
  def __init__(self):
    super().__init__()
    print("Hola desde la clase hijo")

hijo = Hijo()
```

```
hola desde clase padre
Hola desde la clase hijo
```

Problema del diamante

Se llama así porque si hiciéramos un diagrama UML dibujaria un diamante. El problema es identificar cómo es el orden de llamadas cuando tenemos una superclase de las cuales heredan dos clases y a su vez estas son padre de otra clase más.



```python
class SuperSuperClase:
    super_super_llamadas = 0
    def mensaje(self):
      self.super_super_llamadas += 1
      print("llamada a metodo desde SuperSuperClase")

class Superclase1(SuperSuperClase):
  super1_llamadas = 0
  def mensaje(self):
    SuperSuperClase.mensaje(self)
    self.super1_llamadas += 1
    print("llamada a metodo desde SuperClase1")

class Superclase2(SuperSuperClase):
  super2_llamadas = 0
  def mensaje(self):
    SuperSuperClase.mensaje(self)
    self.super2_llamadas += 1
    print("llamada a metodo desde SuperClase2")

class Sublaclase(Superclase1,Superclase2):
  llamadas = 0
  def mensaje(self):
    Superclase1.mensaje(self)
    Superclase2.mensaje(self)
    print("llamada a metodo desde Sublase")
    self.llamadas += 1

mi_clase = Sublaclase()
mi_clase.mensaje()
print(mi_clase.llamadas,mi_clase.super1_llamadas,mi_clase.super2_llamadas, mi_clase.super_super_llamadas)
```

Method Resolution Order (MRO)

Python usa el algoritmo C3 de linearización para determinar el MRO. Define la secuencia en las cuales las clases base son buscadas cuando se busca un método o atributo de un objeto. Importante para evitar ambigüedades y conflictos.

```python
class A:
    def saludo(self):
        print("Hola desde A")

class B(A):
    def saludo(self):
        print("Hola desde B")

class C(A):
    def saludo(self):
        print("Hola desde C")

class D(B, C):
    pass

obj = D()
obj.saludo()
```

```
Hola desde B
```

Mixin

Es una clase que proporciona métodos destinados a ser utilizados como una extensión opcional para otras clases. No están destinados a ser instanciados por sí solos, sino a ser heredados junto con otras clases para proporcionar comportamientos específicos.

- Son útiles para reutilizar funcionalidades en diferentes clases sin que las herencias se vuelvan demasiado complejas.
- Permiten mantener su código modular y DRY (Don't Repeat Yourself) al encapsular comportamientos comunes en clases separadas.

```python
class Log:

  def __init__(self, log_nombre_archivo):
        self.log_nombre_archivo = log_nombre_archivo

  def guardar_en_log(self, mensaje):
    with open(self.log_nombre_archivo, "a") as log_archivo:
            log_archivo.write(f"[LOG] {self.__class__.__name__}: {mensaje}\n")
    print(f"LOG: {self.__class__.__name__} : {mensaje}")

class Database(Log):
  def procesar_consulta(self, consulta):
    self.guardar_en_log("procesa consulta: "+consulta)
    print("aca se trabaja la consulta")

class Servidor(Log):
  def enviar_peticion(self, peticion):
    self.guardar_en_log("envia petición: "+peticion)
    print("aca se hace la peticion al servidor")

bd = Database("database_log.txt")
server = Servidor("servidor_log.txt")

bd.procesar_consulta("select * from mi_tabla")
server.enviar_peticion("https://www.google.com/search?q=python")
```

Polimorfismo

La palabra viene del griego “Polymorphos” Poly=muchos morphos=forma. Del dios morfeo que podía aparecerse en los sueños de las personas de muchas formas.

En biología: las especies que pueden tener diferentes colores o formas (el mismo Homo Sapiens)

En química: los compuestos polimórficos que se pueden cristalizar en al menos dos formas. Ej: Carbón, grafito, diamante

Polimorfismo en lenguajes de programación

El término nació en el paradigma Funcional y después adoptado por Orientado a Objetos.

Básicamente quiere decir **“Un nombre y varios significados”**

- Nombre: de variables, métodos, clases etc..
- Significados: puede variar, al menos se identifican 4 tipos:

1- Overloading, Sobrecarga, Polimorfismo ad-hoc o estático

Cuando el nombre de un método tiene varias implementaciones.

Se distingue en tiempo de compilación, por lo general por los tipos y cantidad de parámetros.

```
class OverLoader {
    // three overloaded meanings for the same name
  public void example (int x) { ... }
  public void example (int x, double y) { ... }
  public void example (string x) { ... }
}
```

Fuente: An Introduction to Object-Oriented Programming. Timothy Budd. Cap.14.1

2- Overriding, Sobreescritura (Polimorfismo por inclusión o dinámico)

Visto en la primer clase, diapositiva 12.

En este caso tenemos el mismo nombre de método, pero la misma “signatura” o protocolo (nombre, cantidad y tipos de parámetros) y además tiene que existir una relación de herencia entre las clases.

3- Variable polimórfica

Es una variable que es declarada de un tipo pero en realidad contiene un valor de otro diferente.

Persona p = new Estudiante() <--- JAVA, en python no declaramos los tipos.

Se declara como la clase padre pero en realidad contiene una instancia de alguna clase hija.

4- Genéricos o templates

Proveen una forma de crear herramientas de propósito general y hacerlas más específicas en ciertas situaciones.

Una función o clase genérica es parametrizada por un tipo , (parecido a como las funciones son parametrizadas con valores). Dejando el tipo sin especificar, para ser llenado luego.

```
template <class T> T max (T left, T right)
{
      // return largest argument
   if (left < right)
      return right;
   return left;
}
```

Fuente: An Introduction to Object-Oriented Programming. Timothy Budd. Cap.14.1

Ver 1er ejemplo en notebook

Duck Typing

"*If it walks like a duck and it quacks like a duck, then it must be a duck*"

Aplicado a la programación, si el objeto sabe responder a los métodos que necesito, es suficiente.

"*Don't check whether it is-a duck: check whether it quacks-like-a duck, walks-like-a duck, etc, etc, depending on exactly what subset of duck-like behavior you need to play your language-games with.*" -- Alex Martelli (miembro de Python Foundation y autor de "Python in a Nutshell")

```python
class Pato:
    def hablar(self):
        print("Cuack Cuack")
class Perro:
    def hablar(self):
        print("Guau")

class Gato:
    def hablar(self):
        print("Miau")

class Gallo:
    def hablar(self):
        print("¡Cocorocooo!")

lista = [Pato(), Perro(), Gato(), Gallo()]
for animal in lista:
    animal.hablar()
```

```
Cuack Cuack
Guau
Miau
¡Cocorocooo!
```

Interfaces

Definen un conjunto de métodos que un objeto debe cumplir, pero no como.

Nos permiten abstraer el “qué” debe hacer, dejando en “cómo” para más tarde, para cuando se tenga que implementar la funcionalidad.

Muy útil para los patrones de diseño.

En python existen las **interfaces informales** y **formales**.

Interfaz informal

Una clase con los métodos definidos pero sin cuerpo y con subclases que le dan la implementación.

```python
class Vehiculo():
    def arrancar(self):
        pass
    def acelerar(self, velocidad):
        pass
    def frenar(self):
        pass
    def apagar(self):
        pass

class Auto(Vehiculo):
    def arrancar(self):
        print("El auto arranca.")

    def acelerar(self, velocidad):
        print(f"El auto acelera a {velocidad} km/h.")

    def frenar(self):
        print("El auto frena.")

    def apagar(self):
        print("El auto se apaga.")
```

Interfaz formal

Una clase abstracta (que hereda de ABC) con los métodos definidos abstractos y con subclases obligadas a darle la implementación.

```python
from abc import ABC, abstractmethod

class Vehiculo(ABC):
    @abstractmethod
    def arrancar(self):
        pass

    @abstractmethod
    def acelerar(self, velocidad):
        pass

    @abstractmethod
    def frenar(self):
        pass

    @abstractmethod
    def apagar(self):
        pass

class Auto(Vehiculo):
    def arrancar(self):
        print("El auto arranca.")

    def acelerar(self, velocidad):
        print(f"El auto acelera a {velocidad} km/h.")

    def frenar(self):
        print("El auto frena.")

    def apagar(self):
        print("El auto se apaga.")
```

Ejercicio

Defina una interfaz Empleado que tenga los métodos calcular_salario() y mostrar_informacion()

Desarrolle la clase EmpleadoTiempoCompleto que tiene como atributos su nombre y su sueldo anual y la clase EmpledoTemporal que tiene nombre, sueldo_por_hora y horas_trabajadas. Ambas deben implementar la interfaz.

Para probar, cree una lista de 3 empleados random e imprima su información.

¿Qué tipo de polimorfismo sucede? ¿estático o dinámico? ¿como se podría implementar el otro tipo?