UNIVERSIDAD
DE MURCIA

Tema 3. Jerarquización

Tecnología de la Programación

Material original por L. Daniel Hernández (*ldaniel@um.es*)
Editado por:
- Sergio López Bernal (*slopez@um.es*)
- Javier Pastor Galindo (*javierpg@um.es*)

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia
27 de septiembre de 2025

Índice de contenidos

Relaciones
- Tipos de relaciones
- Delegación
- Clonación/copia de un objeto
- Ejercicios de relaciones

Herencia
- Concepto y tipos
- Sobreescritura y ocultación
- Problema del diamante
- Polimorfismo
- Uso de la herencia

UNIVERSIDAD
DE MURCIA

Relaciones

Tipos de relaciones

Relaciones entre clases

- Cuando se tiene varias clases, pueden existir vínculos entre ellas.
- A los vínculos se les llama relaciones.
- En su versión más sencilla, un objeto manda un mensaje a otro.
- En su versión más compleja, un objeto necesita la información contenida en otro.
- Distinguimos los siguientes tipos de relaciones de menor a mayor dependencia:
  - Relación de uso
  - Asociación
  - Agregación (has-a)
  - Composición (part-of)
  - Herencia (is-a)
- Estos vínculos establecen una relación jerárquica entre clases.
- Cuándo definir uno u otro tipo de relación dependerá del diseño final (interpretación del programador y principios de ingeniería informática).

Relación de uso

- Una **relación de uso** es una relación en la que una clase utiliza objetos de otra clase, pero es una **relación esporádica**.
- Una clase A usa una clase B para que desarrolle un servicio por él.
- Lo más típico es que se pase una instancia de la clase B a un método de la clase A.

```
class B:
    def usar(self):
        print('Me utilizan')

class A:
    def metodo(self, b: B):
        b.usar()

a: A = A()
b: B = B()
a.metodo(b)
```

- **Ejemplos:**
  - *Las personas usan los cajeros (sin que la persona sea cliente del banco).*
  - *Las personas compran en (se relacionan con) los supermercados.*
  - *Una persona puede usar un servicio público (vehículos, correos, ...).*
  - *etc ...*

Asociación

- La **relación de asociación** representa una conexión lógica entre clases estable en el tiempo. Sin embargo, los objetos relacionados pueden cambiar o dejar de existir.
- No sólo la clase A usa la clase B, sino que un atributo de la clase A es una instancia de la clase B.
- La existencia de un objeto de la clase A no depende de la existencia del objeto de la clase B (el atributo puede quedar nulo).
- La asociación entre dos clases puede ser unidireccional o bidireccional.

```
class B:
    pass

class A:
    def __init__(self, b: B):
        # A mantiene una referencia a B
        self._b: B = b
```

- **Ejemplos:**
  - *Un cliente web depende de un servidor (cambia en el tiempo).*
  - *Los estudiantes se relacionan con los profesores, y a la inversa (cambian en el tiempo).*
  - *Los propietarios de casas se relacionan con sus aseguradoras (cambian en el tiempo).*

Agregación

- Una **relación de agregación** es un caso de asociación donde hay **más nivel de pertenencia** y es menos probable que cambie en el tiempo.
- Se produce cuando una objeto tiene-un objeto | relación 'has-a' |

```
class B:
    pass

class A:
    def __init__(self, b: B):
        # A depende de B
        self._b: B = b
```

- Al igual que en la relación de asociación, la relación no afecta a sus ciclos de vida: aunque A tiene un miembro del tipo B, si destruyes A sigue existiendo B.
- Los objetos pueden existir independientemente.
- **Ejemplos:**
  - *Una factura está asociada a un cliente (y el cliente puede aparecer en varias facturas).*
  - *Una persona puede tener un coche/jersey (y el coche/jersey puede existir sin la persona).*
  - *Una habitación puede tener una o varias sillas (y éstas pueden existir sin la habitación).*
  - *Un estudiante vive en una dirección postal (y ésta existe sin el estudiante).*

Composición

- La **relación de composición** es más restrictiva que la agregación.
- Se produce cuando una objeto es parte-de otro objeto | relación 'part-of' |
- Esta relación **sí afecta a sus ciclos de vida**: un objeto NO puede existir sin el otro.

A tiene un miembro del tipo B. Si destruyes A, también se destruye B.

Además: si B no está en A, el objeto A está incompleto (sin definir).

- El atributo de tipo B suele crearse e inicializarse en el constructor de A
- **Ejemplos:**
  - *Una persona tiene corazón (sin él la persona está incompleta).*
  - *Un rabo forma parte de los perros y gatos (y sin él la mascota está incompleta).*
    Si dejan de existir las mascotas, dejan de existir los rabos.
  - *Una biblioteca tiene libros (pero sin éstos, la biblioteca deja de existir).*
  - *Las habitaciones forman parte de un casa (aunque sea una).*
    Sin habitaciones no tiene sentido que eso sea una casa.
    Si deja de existir la vivienda, dejan de existir las habitaciones.
  - *etc ...*

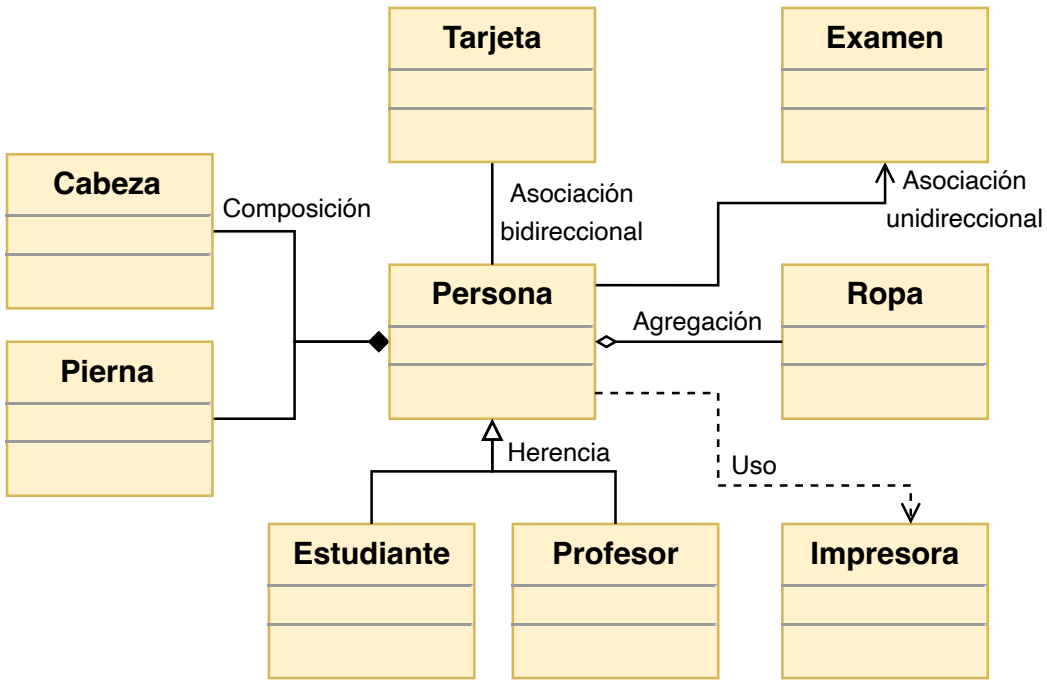
Resumen

- A veces la diferencia entre relaciones no está clara (depende del diseño).
- Estarás en una situación de **relación de uso** cuando un objeto de B no se almacena en ningún atributo de A.
- Si el objeto se almacena en algún campo, un objeto B será un atributo del objeto A.
  - Si no afecta a sus ciclos de vida (destruyes A entonces el objeto B seguirá existiendo):
    - Será una relación de **Agregación** cuando:
      - A tiene o posee otro objeto B, y/o B es parte de A
    - Será una relación de **Asociación** cuando no sea de Agregación.
  - Si sí afectan a sus ciclos de vida (destruyes A entonces el objeto B deja de existir):
    - Será una relación de **Composición**.
- Ejemplo. Imagina un pirata con una pata de palo, espada al cinto y disparando un cañón de su barco pirata:
  - relación de uso con el cañón (no puede llevar el cañón encima)
  - una asociación con la espada (no forma parte del pirata)
  - una pata de palo agregada (sobrevive si matan al pirata)
  - una pierna "normal" y brazos (no sobrevivirán si matan al pirata)

Relaciones en UML

Tipos y significado

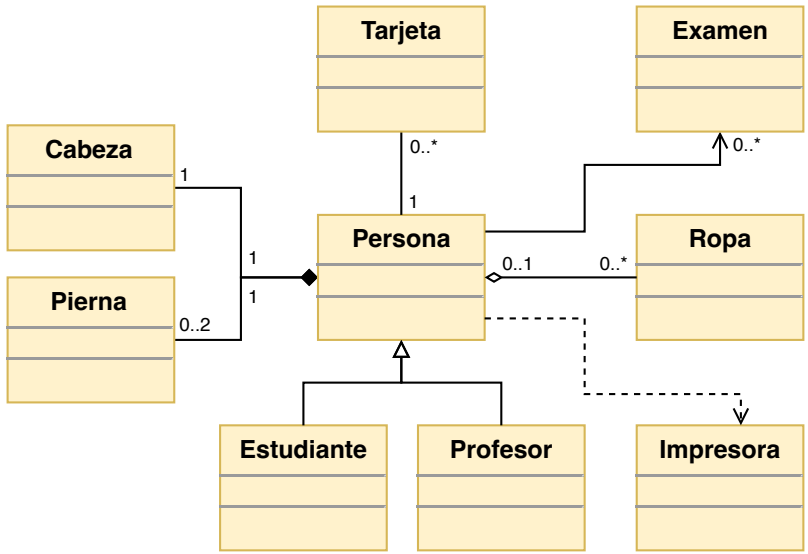
- **Asociación**: Línea continua entre clases. Puede ser unidireccional o bidireccional.
- **Agregación**: Línea continua con rombo blanco en el extremo del "poseedor".
- **Composición**: Línea continua con rombo negro en el extremo del "poseedor".
- **Herencia**: Línea continua con flecha triangular blanca hacia la superclase.
- **Uso**: Línea discontinua con flecha abierta hacia la clase usada.



Relaciones en UML

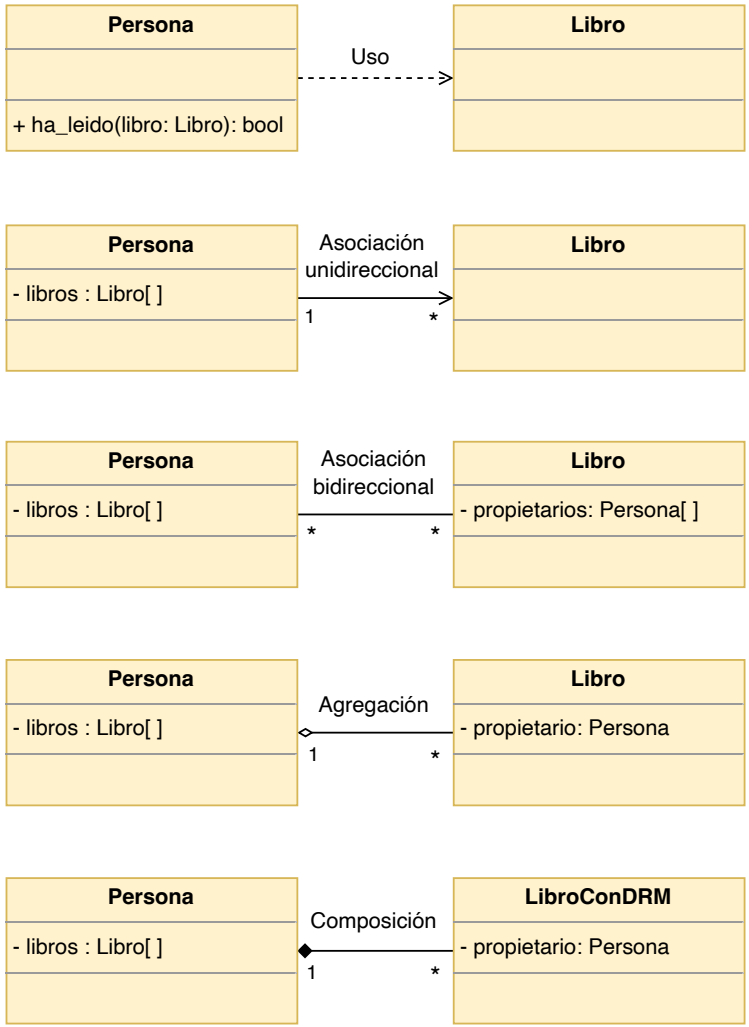
Cardinalidad entre relaciones

- **Cardinalidad**: indica el número de instancias de una clase que pueden asociarse con instancias de otra clase.
- Se coloca en los extremos de las líneas de relación.
- Cardinalidades más comunes:
  - `1` (exactamente una instancia).
  - `0..1` (cero o una instancia).
  - `*` (cero o más instancias).
  - `1..*` (al menos una instancia).
  - `n..m` (rango específico de instancias).
- Ejemplos:
  - Una persona tiene una sola cabeza.
  - Una persona puede tener cero o más tarjetas.
  - Una tarjeta siempre está asociada a una persona.



Ejemplos de relaciones entre Persona y Libro

- **Uso**: Persona tiene un método que usa un objeto de la clase Libro.
- **Asociación**:
  - Unidireccional. Una persona tiene el campo libros que almacena una lista de libros que usa. A su vez un libro lo puede usar muchas personas.
  - Bidireccional. Un libro también tiene un campo con los nombres de las personas.
- **Agregación**. Se considera que el libro tiene como único propietario a una persona. De alguna forma, la persona tiene "su ejemplar".
- **Composición**. Se considera que el libro tiene DRM (es la única persona que puede usarlo).



Agregación vs Composición (con código)

- Observa bien las diferencias.
- La mesa tiene dos referencias, pero cada habitación tiene solo una.

```
class Habitacion:
    ...
    def setMesa(self, mesa: Mesa):
        """
        AGREGACIÓN. La mesa ya existe.
        Sintácticamente igual que ASOCIACIÓN.
        """
        self._mesa: Mesa = mesa;

class Casa:
    def __init__(self, numHabitaciones: int):
        """
        COMPOSICIÓN. Los objetos "part-of" se crean en el constructor
        """
        self._habitaciones: list[Habitacion] = []
        for i in range(0, numHabitaciones):
            self._habitaciones.add(Habitacion())
            # Se crean las habitaciones
```

UNIVERSIDAD
DE MURCIA

Relaciones

Delegación

Delegación: ¡Debe aplicarse siempre!

- Delegación: Cuando una clase contiene una o más instancias de otras clases, entonces la clase **delega** su funcionalidad a los atributos.
- Un objeto recibe una petición y **delega** la ejecución del método a otros objetos.
- Es una buena costumbre que la acción y la acción delegada tengan **el mismo nombre**.

Ejemplo. La clase Rectángulo tiene el método *trasladar* una distancia que, a su vez, delega en el método *trasladar* de la clase Punto que se encarga de modificar las coordenadas *x* e *y* del punto *origen*.

```
class Punto:
    ...
    def trasladar(self, distancia: Punto):
        self.set_x(self.get_x() + distancia.get_x())
        self.set_y(self.get_y() + distancia.get_y())

class Rectangulo:
    def __init__(self, largo: float, ancho: float, origen: Punto):
        self._largo = largo
        self._ancho = ancho
        self._origen = origen

    def trasladar(distancia: Punto):
        self._origen.trasladar(distancia) # Delegación
```

UNIVERSIDAD
DE MURCIA

Relaciones

Clonación/copia de un objeto

¿Cómo clonar un objeto?

- Cuando se tienen dos objetos, la asignación obj1=obj2 produce aliasing sobre el mismo objeto. No se copia el objeto (sino su referencia).
- Hacer una copia conlleva crear una nueva instancia manteniendo las relaciones de asociación, agregación y composición que tiene el objeto.
- **Copia Superficial**
  - Crea una nueva instancia de la misma clase, clonando los valores de los atributos inmutables (numéricos, caracteres, booleanos...) pero referenciando a los atributos mutables (listas, diccionarios, conjuntos y objetos).
  - Es decir, existe **aliasing** en los atributos mutables que quedan compartidos entre los dos objetos clonados.
  - Hay que llevar cuidado, esos atributos con aliasing se pueden modificar a través de ambos objetos.
- **Copia Profunda**
  - Además de la copia superficial, se clonan también los atributos mutables.
  - Se generan dos objetos completamente independientes, sin aliasing.
  - Puede llegar a ser muy compleja.

Clonación de objetos en Python

- El módulo copy permite hacer copias.
- Tiene los siguientes métodos:
  - copy.copy(x) para realizar una copia superficial de x.
  - copy.deepcopy(x[,memo]) para realizar una copia profunda de x.

```
import copy

class Persona:
    def __init__(self, nombre: str, amigos: list[str]) -> None:
        self._nombre: str = nombre  # atributo inmutable (string)
        self._amigos: list[str] = amigos  # atributo mutable (lista)

persona1: Persona = Persona("Juan", ["Carlos", "Ana"])
persona_copia_superf: Persona = copy.copy(persona1) # Copia superficial
persona_copia_superf._amigos.append("Luis") # Modificar lista (ALIASING)
print(persona1._amigos)                  # ["Carlos", "Ana", "Luis"]
print(persona_copia_superf._amigos)  # ["Carlos", "Ana", "Luis"]
# PRUEBA A MODIFICAR EL NOMBRE DE LA PERSONA...

persona_copia_prof = copy.deepcopy(persona1) # Copia profunda
persona_copia_prof._amigos.append("Miguel")
print(persona1._amigos)                # ["Carlos", "Ana", "Luis"]
print(persona_copia_prof._amigos) # ["Carlos", "Ana", "Luis", "Miguel"]
```

- Para que una clase tenga su propia implementación de copy, puede definir métodos mágicos __copy__() y __deepcopy__().

UNIVERSIDAD
DE MURCIA

Relaciones

Ejercicios de relaciones

Ejercicios - I

Ejercicio.

Programa las clases y relaciones necesarias:

- Un usuario tiene dinero en efectivo y una tarjeta. Esta última tiene dueño y saldo. Hay un cajero que permite sacar una cantidad dada de dinero en efectivo con la tarjeta si esta tiene saldo suficiente.
- Una familia se entiende como una lista de personas. Las personas tienen un nombre y pertenecen a una familia, uniéndose a la misma cuando nacen.
- Triángulo cerrado en el plano, definido por tres puntos. En este programa, los puntos sólo pueden formar parte de un triángulo.

Ejercicios - II

Ejercicio.

Si el estado de una persona queda determinado únicamente por su nombre y su pareja:

- ¿cómo se construye que una persona A es soltera?
- Y si A es pareja de B, ¿cómo se pueden construir A y B?

Ejercicios - III

Ejercicio.

Un usuario sabe que está inscrito en una biblioteca y en una tienda de libros. Tanto la biblioteca como la tienda contienen una cantidad ingente de libros y tienen registrado al usuario como cliente. Además, todo libro agrega bien la biblioteca en la que se encuentra depositado o la tienda en la que está disponible.

Programa clases y métodos para que un usuario consulte la disponibilidad de un libro

Ejercicios - IV

Ejercicio.

En un videojuego de acción en tercera persona los jugadores controlan a un personaje. El personaje debe tener siempre su arma, que puede disparar si esta tiene munición. También tiene la capacidad de recoger objetos del entorno del juego que se añadirán a su inventario.

Programa las clases y métodos que permiten simular las acciones de disparar y recoger objetos.

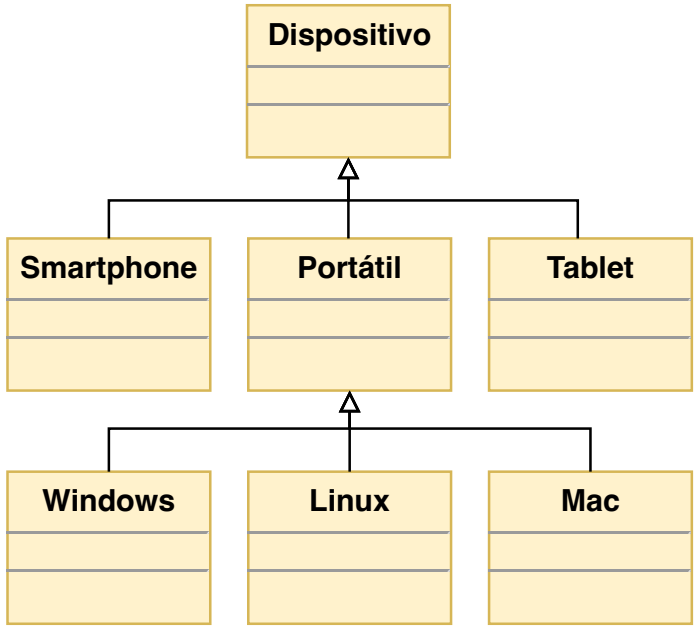
UNIVERSIDAD
DE MURCIA

Herencia

Herencia

- Ya sabes que las clases se pueden relacionar entre ellas por asociación:
  - La asociación de agregación se corresponde con `has-a`.
  - La asociación de composición se corresponde con `part-of`.
- La asociación de **herencia** se corresponde con `Is-a`.
- Se usa cuando una clase es una generalización de otra o, si se prefiere, cuando la otra es un caso particular de la una.

Ejemplo.

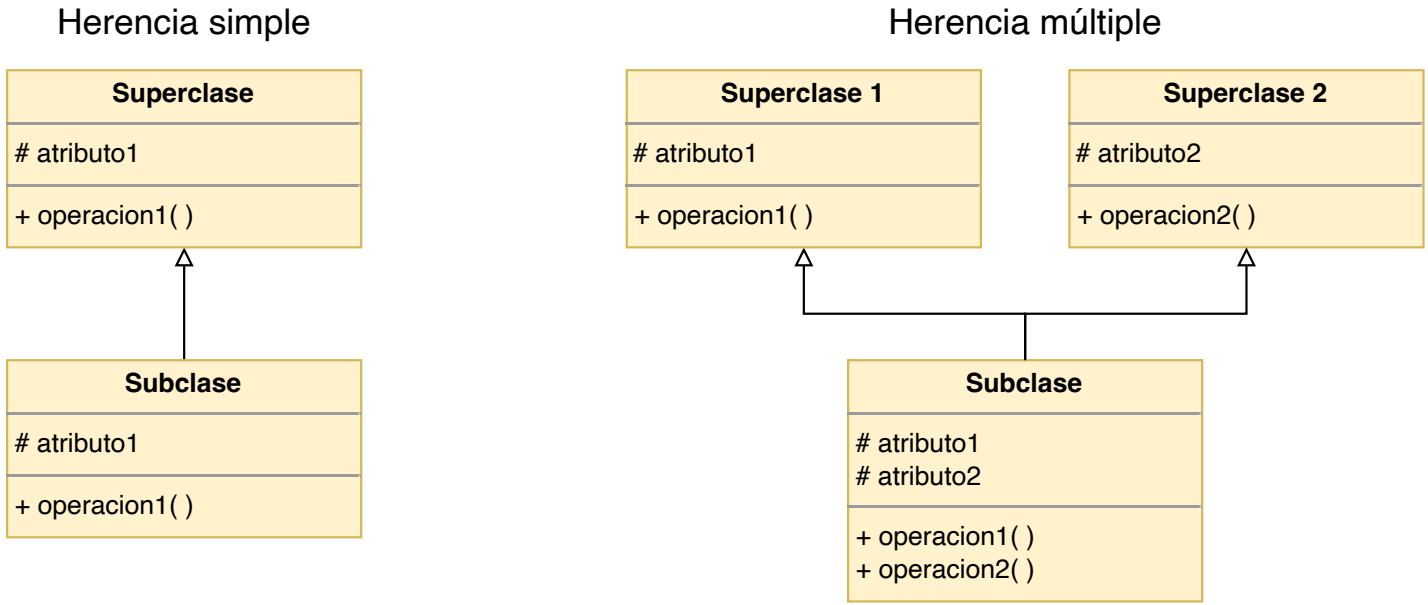


Subclases

- La herencia crea clases nuevas a partir de una clase existente.
  - La clase nueva es un caso particular de la clase que ya existe.
- A la clase nueva se le denomina subclase y a la existente superclase.
  - Responden a una especialización y a una generalización.
- Nombres alternativos
  - Para la **subclase**: clase hija, derivadas o subtipos.
  - Para la **superclase**: clase padre o base.
- La herencia es el proceso por el que una clase hija reconoce a los miembros de la clase padre.
  - Los miembros reconocidos se llaman miembros heredados.
  - No tienen que reconocerse todos (sólo los que se indiquen).
- En la subclase se DEBEN definir nuevos miembros (atributos y/o métodos)
  - Por eso se dice que la clase derivada extiende a la clase base.
  - Los nuevos miembros serán desconocidos por la clase padre.

Tipos de Herencia

- Hay **herencia simple** cuando la subclase solo puede heredar de una clase padre (Java, C#, etc.).
- Hay **herencia múltiple** cuando la subclase hereda de dos o más padres (C++, Python, etc.).



- Un objeto con "tipo de dato" de una subclase también es un "tipo de dato" de las superclases. *Por ejemplo, si Hombre es subclase de Omnívoro y Animal, entonces un objeto de tipo Hombre también es de tipo Omnívoro y Animal.*

Herencia en Python

- En Python la herencia es múltiple.
- La clase hija hereda todos los **miembros públicos y protegidos** de la clase padre. Realmente los privados también, pero se ocultan mediante el *name mangling*.
- Definición de subclases:

  class Subclase (Superclase1, SuperClase2, ...)

- Una subclase puede (debe) llamar al __init__() del padre con super().
  - super() retorna un objeto "proxy"que representa a la clase padre.

```
class Mascota: # Superclase
    def __init__(self, nombre: str):
        self._nombre: str = nombre

    def get_nombre(self) -> str:
        return self._nombre

class Perro(Mascota): # Subclase
    def __init__(self, nombre: str):  # Invoco al constructor padre
        super().__init__(nombre)
        self._nombre = "Perro " + self.get_nombre()  # modificación

perro = Perro("Toby")
print (perro.get_nombre()) # Imprime "Perro Toby"
```

Herencia y Sobreescritura de métodos

- En una superclase se indicará siempre qué miembros serán heredados.
  - Si bien en Python se hereda todo, son los miembros protegidos y públicos de la superclase los únicos que deben ser accedidos.
- Si una subclase hereda un método puede hacer dos cosas:
  - Usar el método de la superclase como si fuera suyo.
  - Sobreescribir (*Overriding*) el método para tener un comportamiento diferente (pudiendo usar super() para "extender" el método padre).

La sobreescritura de un método es construir un nuevo método con la misma declaración (signatura) pero donde cambia la definición. Si no se sobreescribe, se mantiene la definición del método padre.

```
class Mascota: # Una clase
    def __init__(self, nombre: str):
        self._nombre: str = nombre

    def get_nombre(self) -> str:
        return self._nombre

class Perro(Mascota): # Subclase, constructor se hereda si no se redefine

    def get_nombre(self) -> str: # sobreescritura de get_nombre
        return f"Perro se llama {self._nombre}" # redefinición completa
        return f"El perro se llama {super().get_nombre()}" # o extensión
```

Herencia y Ocultación de atributos

- Si una subclase hereda un atributo puede hacer dos cosas:
  - Usar el atributo de la superclase como si fuera suyo.
  - Ocultar el atributo heredado mediante una nueva definición.
    La ocultación de un atributo es definir una atributo con el mismo identificador que el atributo de la superclase. Si no se oculta/redefine, se mantiene el valor del atributo padre.
- Un atributo se comparte hacia abajo por la jerarquía de herencia. En el constructor, con super() podemos heredar los atributos de la clase padre.

```
class Mascota:
    def __init__(self, sonido: str = "Sonido genérico"):
        self._sonido: str = sonido

class Perro(Mascota):
    def __init__(self, tipo: str):
        super().__init__(sonido="Guau") # Usar el atributo padre sonido
        # self._sonido = "Guauguau"     # Ocultaría el atributo padre
        self._tipo = tipo             # Nuevo atributo, solo en subclase

perro = Perro(tipo="Chihuahua")
print(perro._sonido)     # Imprime "Guau"
print(perro._tipo)       # Imprime "Chihuaua"

animal = Mascota()       # No tiene tipo
print(animal._sonido)    # Imprime "Sonido genérico"
```

Relación entre clases y subclases

- RECUERDA: Una subclase es la particularización de la clase padre.
- La **subclase reconoce a los miembros de la superclase** y añade (o redefine) nuevos miembros.
  - Para poder acceder a los métodos de la superclase se deberá usar la función super().
  - Es común que en la **sobreescritura de un método** `metodo(...)`, la primera instrucción invoque al método padre `super().metodo(...)` y las siguientes instrucciones modifiquen o extiendan con nueva funcionalidad lo definido en la clase padre.
  - Es común que en la **ocultación de atributos**, la clase hija `__init__(...)` invoque al constructor padre `super().__init__(...)` para inicializar los atributos heredados y después extender con nuevos atributos (u ocultar los heredados). Es decir, cada __init__():
    - pase a super() los argumentos que no son propios,
    - se creen nuevos atributos con el resto de argumentos.
- La **clase padre no puede acceder a los miembros de la subclase**.
  - obj.__dict__ es un diccionario que muestra los miembros propios.
  - `dir`(obj) es una lista con los miembros del objeto, tanto propios como recursivamente heredados.

Relación entre clases y subclases

Ejemplo

```python
class Mascota: # Una clase

    def __init__(self, nombre: str):
        self._nombre: str = nombre

    def get_nombre(self) -> str:
        return self._nombre

class Perro(Mascota): # Subclase sin constructor, usará el de Mascota
    cardinal: int = 0

perro = Perro("Toby")       # en Mascota se asigna "Toby" a self._nombre
print(perro.get_nombre())   # se muestra el valor de self._nombre

### Miembros únicos en las clases y el objeto perro
print([ m for m in Mascota.__dict__ if not m.startswith('__')],
      [ m for m in Perro.__dict__ if not m.startswith('__')],
      [ m for m in perro.__dict__ if not m.startswith('__')])
# ['get_nombre'] ['cardinal'] ['_nombre']

### Miembros reconocidos en las clases y el objeto perro
print([ m for m in dir(Mascota) if not m.startswith('__')],
      [ m for m in dir(Perro) if not m.startswith('__')],
      [ m for m in dir(perro) if not m.startswith('__')])
# ['get_nombre'] ['cardinal', 'get_nombre']
# ['_nombre', 'cardinal', 'get_nombre']
```

Relación entre clases y subclases

Un ejemplo más claro donde los atributos se añaden al objeto

```python
class Mascota: # Una clase

    def __init__(self, nombre: str):
        self._nombre: str = nombre

    def get_nombre(self) -> str:
        return self._nombre

class Perro(Mascota): # Subclase

    def __init__(self, nombre):

        # Añade atributo _nombre
        super().__init__(nombre: str)

        # Modifica atributo _nombre
        self._nombre = "EL " + nombre

mascota: Mascota = Mascota("Toby")
print(mascota.get_nombre()) # Toby

perro: Perro = Perro("Toby")
print(perro.get_nombre()) # EL Toby
```

El atributo **protegido** self._nombre es compartido en las dos clases.

```python
class Mascota: # Una clase

    def __init__(self, nombre: str):
        self.__nombre: str = nombre

    def get_nombre(self) -> str:
        return self.__nombre

class Perro(Mascota): # Subclase

    def __init__(self, nombre):

        # Añade atributo en el padre
        super().__init__(nombre)

        # Nuevo atributo
        self.__nombre = "EL "+nombre

perro: Perro = Perro("Toby")
print(perro.get_nombre()) # Toby

# Equivale a lo siguiente (NO HACER)
print(perro._Mascota__nombre)
```

Hay un atributo **privado** __nombre en cada clase. Habría que sobreescribir el método get_nombre().

Sobreescritura sobre la clase *Object*

- En Python realmente hay una clase raíz Object de la que heredan implícitamente todas las clases existentes o que definimos.
- Cuando implementamos métodos mágicos, realmente estamos sobreescribiendo métodos ya definidos en la clase raíz Object:
  - *__init__ ()*. El método de inicialización de variables de un objeto.
  - *__str__ ()*. El modo en el que un usuario vería la información de la clase. Retorna una cadena "informal".
  - *__gt__ ()*, *__ge__ ()*, *__lt__ ()*, *__le__ ()*: Métodos que define las desigualdades lógicas al comparar dos objetos con $>$, $>=$, $<$ y $<=$.
  - __eq__(self , objeto). Método que define el operador de igualdad ($==$)
  - etc...
- Todos estos métodos ya tienen una implementación previa en la clase padre Object, lo que hacemos en los métodos mágicos es sobreescribirlos para personalizar su funcionamiento.

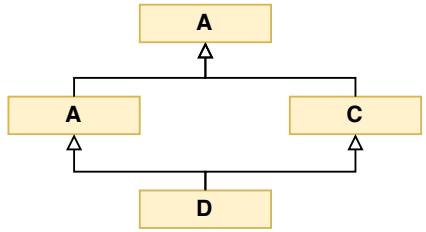
UNIVERSIDAD
DE MURCIA

Herencia

Problema del diamante

Sobreescritura: Problema del diamante

- En herencia múltiple surge el problema de la estructura del diamante.



- Si B y C heredan y sobreescriben un método de A, y la clase D lo hereda (sin sobreescribir) de B y de C, ¿la clase D utiliza el método heredado de B o de C?
- Python usa el Method Resolution Order (MRO): crea una lista recursiva de clases, de izquierda a derecha y de abajo a arriba (D, B, A, C, A), eliminando las clases repetidas salvo la última ocurrencia[1]:
  - El orden es aquel especificado en la definición de la subclase (class Subclase (Superclase1, SuperClase2, ...))
  - La clase raíz siempre será Object.
  - El **orden de resolución** del método es: D, B, ~~A~~, C, A, Object.
- El árbol de ancestros puede obtenerse con el atributo de clase __mro__. Si un método no está en la superclase, se pasa a la siguiente (__mro__).

[1] Para no visitar antes la superclase (A) de una subclase (C)

Problema del diamante

Method Resolution Order (MRO)

Python resuelve super() siguiendo el árbol de ancestros definido por __mro__ del **objeto invocador**.

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

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

class C(A):
    def __init__(self):
        print("C")
        super().__init__()

class D(B,C):
    def __init__(self):
        print("D")
        super().__init__()

a = A()   # imprime A
b = B()   # imprime B, A
c = C()   # imprime C, A
d = D()   # imprime D, B, C, A
```

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

class B(A):
    pass

class C(A):
    def __init__(self):
        print("C")

class D(B, C):
    def __init__(self):
        print("D")
        super().__init__()

a = A()   # imprime A
b = B()   # imprime A
c = C()   # imprime C
d = D()   # imprime D, C
```

- ¿Qué pasaría si C tampoco tuviera __init__()?
  D imprimiría D y A.
- ¿Puede usar D el método de A sin pasar por B/C?
  Sí, usando super(C, self).metodo() o A.metodo(self).metodo(). **NO HACER.**

UNIVERSIDAD
DE MURCIA

Herencia

Polimorfismo

Polimorfismo

- Dada una clase padre y dos clases hijas, éstas pueden heredar un método que pueden sobreescribir (para reemplazarlo o refinarlo).
- Cada clase tiene el *mismo método*, pero lo ejecutan de diferente forma.
  - No confundir con la sobrecarga, donde un mismo método puede tener diferentes parámetros para comportamientos diferentes.
- El Polimorfismo es la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de clases diferentes.

```
class Animal:
    def sonido(self):
        return "El animal hace ruido"

class Perro(Animal):
    def sonido(self):
        return "El perro ladra" # reemplazo

class Gato(Animal):
    def sonido(self):
        return "El gato maúlla" # reemplazo

animales: list[Animal] = [Perro(), Gato()]

for animal in animales:
    print(animal.sonido()) # poliformismo del método sonido
```

Métodos para identificar la clase o subclase en Python

Un objeto no pertenece a varias clases, pero se comporta como si fuese también del tipo de las clases predecesoras.

- `type(obj)`. Devuelve la **clase exacta** de un objeto.

```python
class Animal:
    pass

class Perro(Animal):
    pass

mi_perro: Perro = Perro()
print(type(mi_perro) == Perro)  # True
print(type(mi_perro) == Animal) # False
```

- `isinstance(obj, clase)`. Verifica si un objeto es **instancia de una clase o subclase**.

```python
print(isinstance(mi_perro, Perro))    # True
print(isinstance(mi_perro, Animal))   # True
```

- `issubclass(clase1, clase2)`. Verifica si **una clase es subclase de otra**.

```python
print(issubclass(Perro, Animal))   # True
print(issubclass(Perro, Gato))     # False
```

UNIVERSIDAD
DE MURCIA

Herencia

Uso de la herencia

Cuándo usar herencia

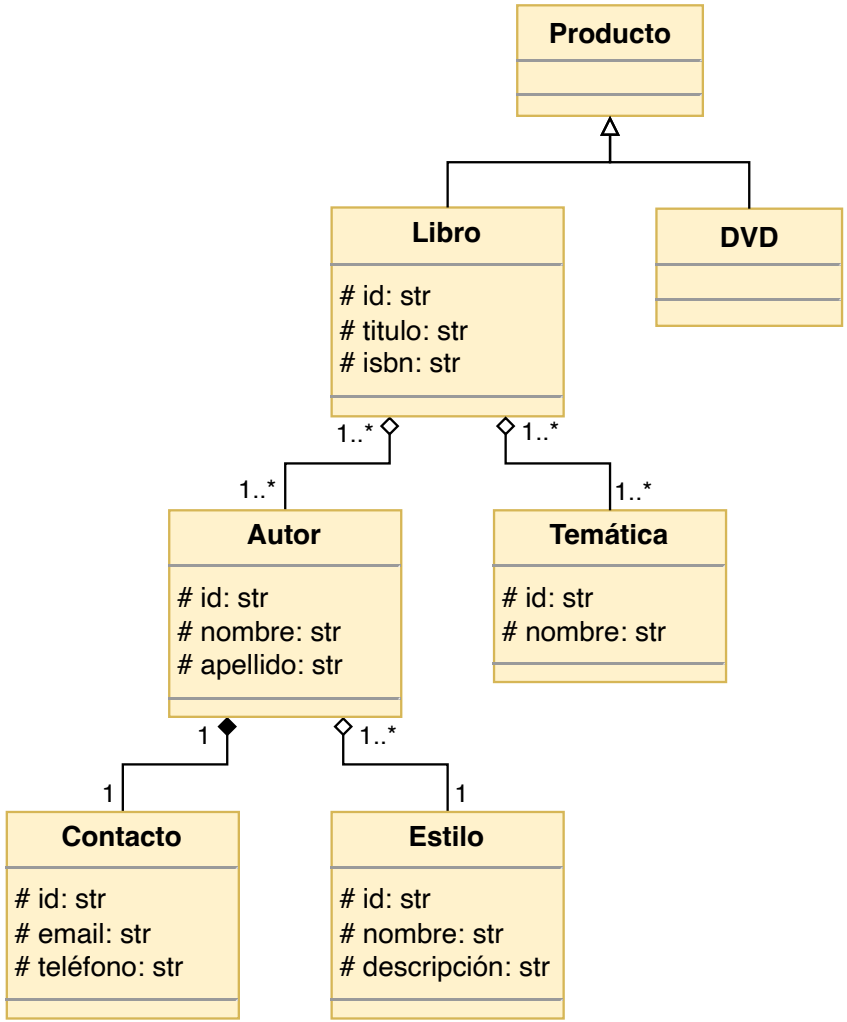
- Usa herencia cuando
  - Relaciones ES-UN. Si se rompen por algún motivo... ¡Mal asunto!
  - TODOS los métodos públicos de la clase A lo son también de la clase B: (1) La subclase B está siempre basada en la superclase A y (2) la implementación de la superclase A es necesaria para B.
  - La subclase es candidata a
    - sólo añadir nueva funcionalidad (nuevos métodos/atributos).
    - no sobrescribir nada: dejarlo como está.
- Principios **SOLID**:
  - **S - Responsabilidad Única:** Cada clase debe tener una sola función o responsabilidad.
  - **O - Abierto/Cerrado:** Puedes extender el comportamiento de las clases sin modificar el código existente.
  - **L - Sustitución de Liskov:** Las subclases deben poder reemplazar a la clase base sin alterar la funcionalidad.
  - **I - Segregación de Interfaces:** Las clases deben implementar solo los métodos que necesitan y no depender de métodos innecesarios.
  - **D - Inversión de Dependencia:** Las superclases no deben depender de detalles específicos de implementación (importancia de clases genéricas, abstractas o interfaces). La lógica específica debe ir en las subclases.

Cuándo NO usar herencia

- **No uses herencia** cuando cause más desventajas que beneficios:
  - Cuando el acoplamiento entre clases es demasiado fuerte.
  - Cuando las clases base crecen y requieren constantes modificaciones (un pequeño cambio afecta a muchas clases).
  - Cuando las subclases sobrescriben demasiados métodos (esto puede indicar que no se trata de una relación de herencia).
  - Cuando las subclases heredan métodos que no necesitan.
  - Cuando la superclase es heredada por solo una subclase.
- **Alternativa**: Asociación, agregación o composición:
  - En lugar de que B herede de A, A puede estar asociada a B. Esto es una relación diferente con cambios de diseño importantes.
  - Modelan una relación con menor acoplamiento.
  - Los cambios en una clase afectan mínimamente a otras clases.
  - No tendrás beneficios como la reutilización o polimorfismo.
- ¿Qué pasa si quiero **objetos del mismo tipo pero sin acoplamiento**?
  ¡Usa interfaces! Lo explicaremos en el próximo tema.

Ejemplo con diferentes tipos de relaciones entre clases

Los libros y los DVDs son productos culturales.
En concreto, los libros tienen un título y un ISBN. Están asociados a un conjunto de temáticas (una temática puede estar asociada a muchos libros). Y también están asociados a un listado de autores (un autor podría asociarse a otros libros).
Cada autor tiene un nombre y su apellido. Se compone con sus datos de contacto (un email y un teléfono), que se destruyen cuando el autor muere. Un autor se asocia con un estilo (que puede ser utilizado por muchos autores). De un estilo se guarda el nombre y la descripción.



Ejercicios

Ejercicio.

En un sistema de simulación están los **agentes** que permiten **decidir** y aquellos que además se pueden **mover**.

Ejercicio.

Todo **cliente** se caracteriza por tener un DNI y una cuenta bancaria, que puede ser de ahorro o de crédito. Si es de ahorro, genera unos intereses; pero si es de crédito permite tener un depósito. Una cuenta bancaria se crea con un saldo inicial. En toda cuenta se puede depositar y retirar dinero. Un DNI consta de un identificador junto con el nombre, dirección y edad al que pertenece.

Ejercicios

Ejercicio.

Toda **alarma** tiene un **umbral** de sensibilidad de intrusos. También consta de un **sensor** al que consulta y que le indica cual es el valor actual de intrusión. En el caso de que se supere el umbral, puede ocurrir lo siguiente.
Si es una alarma **sonora**, pondrá en marcha un timbre incorporado que se puede activar o desactivar. Pero si es una alarma **luminosa**, encenderá una luz. En el caso de que sea **sonora y luminosa** hará las dos cosas.

Ejercicio.

Para el ejercicio anterior de la alarma ¿qué modificaciones tendrías que hacer para que una alarma completa encendiera la luz ante cierto nivel de intrusión, pero que hiciera sonar también el timbre si el nivel fuera aún mayor?