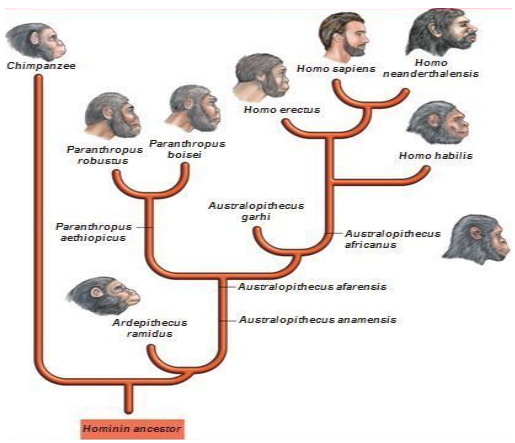
UNIVERSIDAD DE MURCIA

Jerarquización

Composición vs Herencia

L. Daniel Hernández < *ldaniel@um.es* >

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia
13 de noviembre de 2023



1

[1] Imagen: Evolución Humana: https://images.app.goo.gl/BisXt9wS8na9vfgP9

Índice de Contenidos

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Desarrollo

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Profundizamos en ...

Lluis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Relaciones entre clases

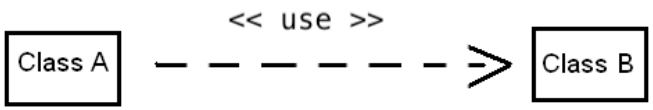
- Cuando se tiene varios tipos de objetos, pueden existir vínculos entre ellos.
- A los vínculos se llaman **relaciones**.
- En su versión más sencilla un objeto simplemente manda un mensaje a otro.
- En su versión más compleja un objeto usa toda la información contenida en otro
- Distinguimos los siguientes tipos de relaciones de menor a mayor dependencia:
  - Relación de uso
  - Asociación
  - Agregación (has-a)
  - Composición (part-of)
  - Herencia (is-a)
- Estas relaciones establecen una relación jerárquica (no estricta) entre clases.
- **Cuándo definir uno u otro tipo de relación dependerá del diseño final (interpretación del programador y principios de ingeniería informática).**

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

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**. Es la relación más general posible entre dos clases.

- Para indicar que una clase A usa una clase B, se representa por:




```
>>> class B:
...     def use(self):
...         print('me utilizan')
...
>>> class A:
...     def metodo(self, b):
...         b.use()
...
>>> a = A(); b = B(); a.metodo(b)
me utilizan
```

- Lo más típico es que se pase una instancia de la clase B a un método de la clase A.

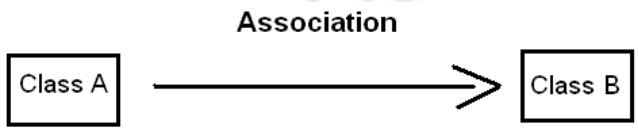
- **Ejemplos:**
  - *Las personas usan los cajeros (sin que la persona sea cliente del banco)*
  - *Las personas compran en (se relacionan con) los supermercados*
  - *Una persona puede usar un servicio público (vehículos, correos, ayudas, ...)*
  - *etc ...*

L. Daniel Hernández · Dpto. Ingeniería de la Información y las Comunicaciones · ldaniel@um.es

Relación Asociación

- La **relación de asociación** es una **relación de uso** (un objeto le requiere a otro objeto que desarrolle un servicio por él – paso de mensajes), **y además** se quiere que sea una relación de dependencia **estable en el tiempo**.

- Para indicar que una clase A está asociada (depende de) una clase B, se representa por:




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

- Un atributo de A es una instancia de B
- La existencia de As no depende de la existencia de Bs (el atributo puede quedar nulo)

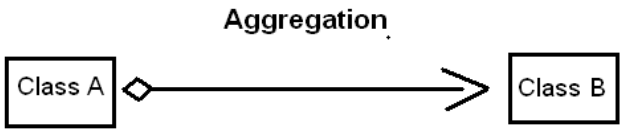
- **Ejemplos:**
  - *Un cliente web depende de un servidor (cambia en el tiempo)*
  - *Los estudiantes se relacionan con los profesores, y a la inversa (cambian en el tiempo).*
  - *Los propietarios de casas se relacionan con sus aseguradoras (cambian en el tiempo).*
  - *etc ...*

Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Agregación

- Una **relación de agregación** es un caso particular de asociación.
- Se produce cuando una objeto **tiene-un** objeto | relación 'has-a' |

- Para indicar que una clase A tiene-un objeto de B y/o B es parte de A se representa por:

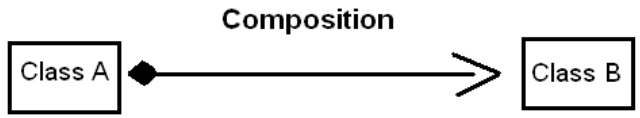



```
>>> class B:
...     pass
...
>>> class A:
...     def __init__(self, b):
...         # A depende de B
...         self.__b = b
...
```

- 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 asociado 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)*
  - *Una biblioteca tiene estudiantes (y sin éstos, la biblioteca existe)*
  - *etc ...*

Composición

- La **relación de composición** es más restrictiva que la agregación (relación 'has-a').
- Se produce cuando una objeto es **parte-de** otro objeto | relación 'part-of' |
- Además la 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).

- **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, deja de existir las habitaciones.
  - *etc ...*

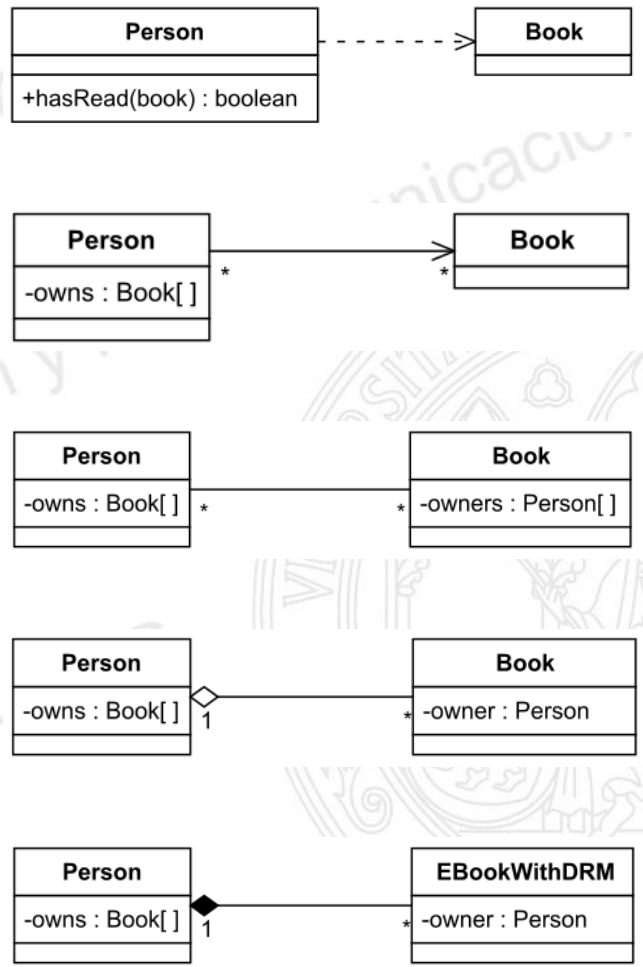
Resumen

- A veces la diferencia entre las 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 campo 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 **Asociación** cuando no sea de Agregación.
    - Será una relación de **Agregación** cuando:
      - A tiene o posee otro objeto B,
      - y/o B es parte de A
  - Si sí afectan a sus ciclos de vida (destruyes A entonces el objeto B deja de existir):
    - Será una relación de **Composición**.
    - Normalmente cuando la relación es de **agregación con aridad 1 a 1**.
    - En Java se recomienda usar la palabra `final` para el atributo. Fuerza a que el objeto "propietario" no puede ser creado si no se inicializa con el objeto miembro (que ya debe de existir).
- Ejemplo: Imagina un pirata con una pata de palo, espada al cinto y disparando un cañón de su barco pirata. El pirata tiene las siguientes relaciones:
  - relación de uso con el cañón (no puede llevar el cañón encima)
  - una asociación con la espada (podría llevar encima otra arma como una pistola)
  - una pata de palo agregada (es parte del pirata, pero sobrevive si matan al pirata)
  - se compone de una pierna "normal" y brazos (son parte del pirata que no sobrevivirán si matan al pirata)

Resumen (con ejemplo)

Fuente: http://www.cs.utsa.edu/~cs3443/uml/uml.html

- **Dependencia**: Persona tiene un método que usa un objeto de la clase Libro.
- **Asociación**:
  - **Unidireccional**. Una persona tiene el campo `owns` 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). Si la persona muere nadie puede usarlos.



En https://www.guru99.com/association-aggregation-composition-difference.html tiene una lectura muy breve que compara asociación, agregación y composición.

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):
        """
        AGREGACIÓN. La mesa ya existe.
        Sintácticamente igual que ASOCIACIÓN.
        """
        self._mesa = mesa;

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

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Profundizamos en ...

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

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 docencia en la universidad se delega a los centros, los centros a los departamentos y los departamentos a los profesores.

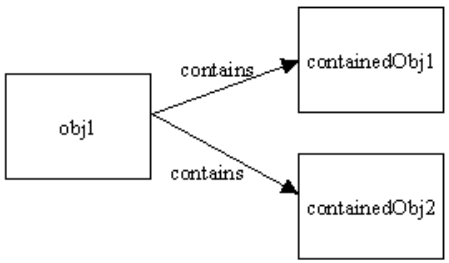
```
class Punto:
    def __init__(self, x: float, y: float):
       pass
    def trasladar(distancia: Punto):
       self._x += distancia.x  # Propiedades
       self._y += distancia.y

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 = sel._origen.trasladar(distancia) # Delegación
```

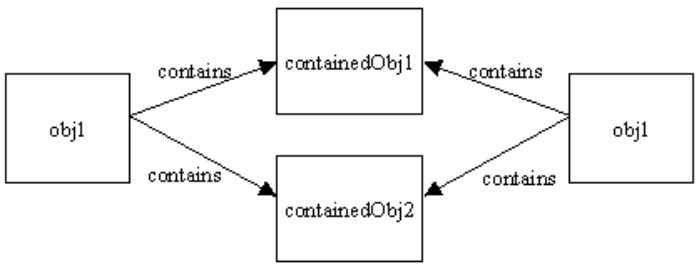
Profundizamos en ...

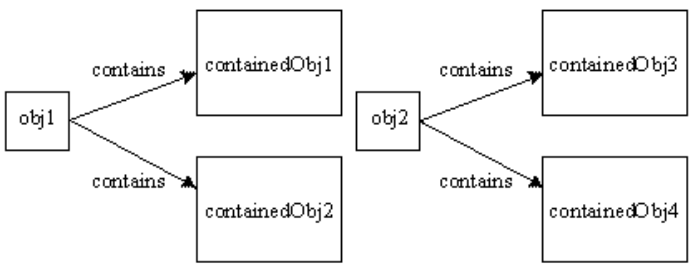
Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

¿Cómo clonar un objeto?

- Cuando se tienen dos objetos, `obj1` y `obj2`, la asignación `obj1=obj2` produce aliasing sobre el mismo objeto. En ningún caso hace una copia del objeto.
- Hacer una copia conlleva mantener las relaciones de asociación, agregación y composición que tiene el objeto original, si las hubiera.
  La tarea no siempre es trivial.







- **Copia Superficial**
  - Generar una instancia de la misma clase y copiar los valores de los atributos.
  - ¡¡ Puede existir aliasing en los atributos !!
- **Copia Profunda**
  - Además de la copia superficial realiza una copia de todos los objetos a los que se refiera.
  - Puede llegar a ser muy compleja.
- Existen **otras formas** de copiar objetos:
  - Defensive copying, Copy constructors, Factory methods.

Clonación de objetos en Python

- El módulo `copy` permite hacer copias.
- Usa las funciones del módulo `pickle` para la (de-)serilización de objetos.
- 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`.
    Para evitar la copia recursiva o la copia de demasiados objetos
    - Mantiene un diccionario `memo` de los objetos ya copiados.
    - Permite al usuario redefinir los operadores de copia.
- Para que una clase defina su propia implementación de `copy`, puede definir métodos especiales `__copy__()` y `__deepcopy__()`.
  Detalles para la implementación profunda ver https://docs.python.org/3/library/copy.html
- Hay funciones que realizan copias:
  - `list()` crea una lista nueva a partir de su argumento (si puede)
  - `dict.copy()` crea una copia superficial del diccionario `dict`

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Ejercicios - I

Ejercicio.

Realiza el diagrama completo de las siguientes relaciones que existe entre estas clases, indicando si es asociación, has-a o part-of:

- Una universidad y los departamentos.
- Un departamento y los profesores.
- Los profesores y los alumnos.
- Los alumnos y un grado.
- Un grado y las asignaturas del grado.
- Las asignaturas y los profesores.

Ejercicio.

Modela la siguiente situación. Todos los piratas tienen corazón pero su número de piernas puede ir de 0 a 2. Además alguna pierna puede ser una pata de palo. Explica el tipo de asociación y como se pondría de relieve con el correspondiente código.

Ejercicios - II

Ejercicio.

Representa y codifica las siguientes relaciones:

- Un usuario saca dinero de un cajero con una tarjeta, si es válida.
- Las personas se añaden a una familia y la familia se actualiza con las personas. Una familia se entiende como una lista de personas.
- Polígono cerrado en el plano, entendiendo como tal a una lista de puntos.

Ejercicios - III

Ejercicio.

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

- ¿cómo se construye que una persona A es soltera?
- ¿y si tiene pareja? Si A es pareja de B, ¿cómo se pueden construir A y B?
- Indica el tipo de variables que intervienen en todo el proceso.

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 tienen registrado al usuario como cliente. Además todo libro agrega la biblioteca en la que se encuentra depositado y la tienda en la que fue comprado. ¿Qué debería de hacer el usuario para consultar la disponibilidad de un libro?

Ejercicios - IV

Ejercicio.

En un videojuego de acción en tercera persona los jugadores controlan a un personaje. El personaje tiene la capacidad de disparar; es decir puede realizar ciertas acciones entre las que se encuentra la de transportar un arma que puede ser disparada. También tiene la capacidad de recoger objetos del entorno del juego; es decir puede interactuar con los objetos y en particular tiene un inventario en el que se pueden añadir objetos. ¿Cuáles son las clases y métodos que permiten similar las acciones de disparar y recoger objetos?

Dpto. Ingeniería... ldaniel@um...

Desarrollo

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Profundizamos en ...

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Introducción a la Herencia

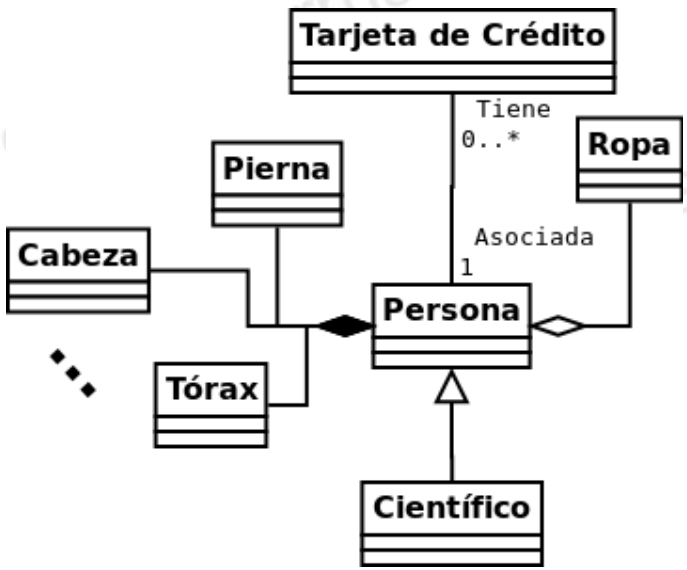
- Las **clases de objetos** pueden verse como **conjunto de elementos**, cada uno caracterizado por su estado y su comportamiento.
- Como conjuntos, sus elementos pueden compartirse o no.
- Matemáticamente, si se comparte se habla de **intersección**.
  - La intersección de dos conjuntos es el conjunto formado por los elementos comunes.
- Cuando la intersección entre dos conjuntos es no vacía entonces puede ocurrir
  - Solo se comparten algunos elementos.
  - Un conjunto comparte todos los elementos con otro.
    En este caso se dice que el primero está incluido en el segundo.
- Este tema va de **inclusiones de clases**.
- En POO a la inclusión se le llama **herencia**.

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Herencia

- Ya sabe que las clases se pueden relacionar entre ellas por asociación. En particular
  - La asociación de **agregación** se corresponde con las relación `has-a`
  - La asociación de **composición** se corresponde con las relación `part-of`.
- La asociación entre clases de **herencia** se corresponde con la relación `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:



Fuente: http://fundamentospoorrr.blogspot.com/2019/03/composicion-agregacion-y-asociacion.html

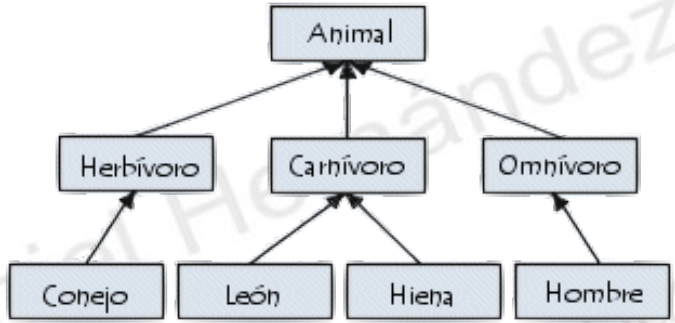
Subclases

- 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**, respectivamente.
- **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**
  - En principio, no tiene que reconocer a 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.

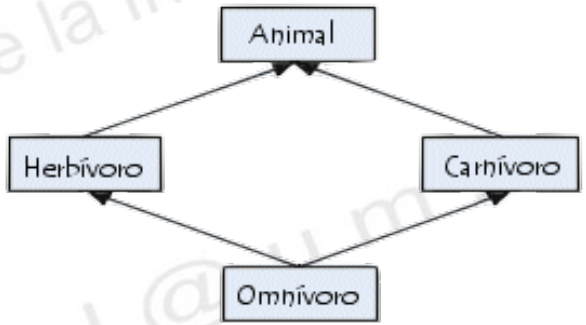
Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Tipos de Herencia

- Hay **herencia simple** cuando la subclase solo puede heredar de una clase padre. Visualmente, el gráfico UML tiene forma de árbol.



- Hay **herencia múltiple** cuando la subclase hereda de dos o más padres. Visualmente, el gráfico tiene forma de grafo.



- Un objeto con "tipo de dato" de una subclase también es un "tipo de dato" de una superclase de la misma rama. P.e. un objeto de tipo hombre también es de tipo omnívoro y animal.

Fuente: https://es.ccm.net/contents/411-poo-herencia

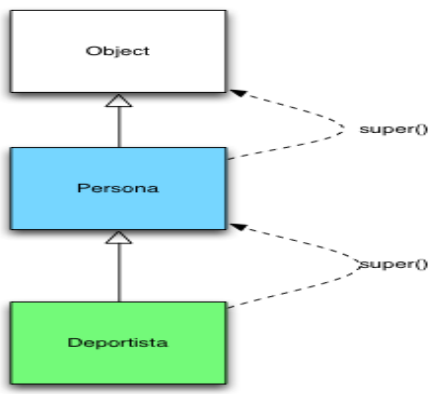
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Herencia en Java

- En Java **la herencia es simple** y no admite herencia múltiple.
- El modificador `protected` es un modificador específico para la herencia de miembros.
- Con éste, ya tenemos todos los modificadores de visualización de Java:

Modificador	Clase	Package	Subclase	Todos
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
No especificado (friendly)	Sí	Sí	No	No
private	Sí	No	No	No

- Definición de subclases: `class Subclase extends Superclase {...}`
- `extends` se usa para generar una subclase (especialización) de un tipo de objeto.
- Una subclase puede (debe) llamar al constructor del padre con `super()`.
  - `super()` representa al constructor de la clase padre, y se llamará con tantos argumentos como requiera dicho constructor.
  - `super()` debe ser siempre la primera línea de un constructor (y excluye a `this()` – uno u otro)



Herencia en Python

- En Python **la herencia es múltiple**.
- Se heredan todos los miembros (no existen modificadores específicos para herencia)
- Definición de subclases:

```
class Subclase (Superclase1, SuperClase2, ...)
```

- En herencia múltiple surge **el problema de la estructura del diamante**: Si `B1` y `B2` heradan y sobreescriben un método de `A` y la clase `C` no lo sobreescribe pero hereda de `B1` y de `B2`, entonces el método heredado en `C` ...

... ¿hay herencia repetida? ¿es de `B1` o es de `B2`? Siempre usa la primera superclase

- `Python` usa el **Method Resolution Order** (MRO): crea una lista de clases que se buscan de izquierda a derecha y de abajo a arriba (`C`, `B1`, `A`, `B2`, `A`) y luego elimina todas las apariciones de una clase repetida menos la última.
  - Por tanto, el orden de resolución del método es: (`C`, `B1`, `B2`, `A`) .
  - La clase raíz siempre es `object`.
- El árbol de ancestros puede obtenerse con el atributo `__mro__`.
- Una subclase puede (debe) llamar al `__init__()` del padre con `super()`.
  - `super()` es una referencia indirecta calculada.
  - Retorna un **objeto proxy**: un objeto que delega las llamadas a los métodos correctos sin crear un objeto adicional.
  - El cálculo depende tanto de la clase desde donde `super()` es llamada como del árbol de ancestros de la instancia.

Profundizamos en ...

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Sobreescritura y Ocultación

- En una superclase se indicará siempre **qué miembros serán heredados**.
  - En `Python`, se heredan todos.
- 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 que el comportamiento sea diferente
    La **sobreescritura** de un método es construir un nuevo método con la misma declaración (signatura) pero donde cambia la definición.
- Si una subclase **hereda una variable** puede hacer dos cosas
  - **Usar** la variable de la superclase como si fuera suyo
  - **Ocultar** la variable heredada
    La **ocultación** de una variable es definir una variable con el mismo identificador de la variable de la superclase.
    - **Dependiendo del lenguaje de programación:** para que realmente haya ocultación también debe coincidir el tipo de dato.
- **Cuando se sobrescriben/ocultan miembros** (métodos/variables) en una subclase, el comportamiento no es exactamente el mismo en distintos lenguajes de programación

Sobreescritura y Ocultación en Java y Python

- **Cuando se sobrescriben/ocultan miembros** (métodos/variables) en una subclase, el comportamiento no es exactamente el mismo en distintos lenguajes de programación
- Java
  - **No significa que se destruyan** los miembros en la superclase.
  - Los miembros sobreescritos de la superclase **estarán ocultos**.
  - Cuando en la subclase se haga **referencia a los miembros sobreescritos** **siempre se hará referencia a las nuevas definiciones**.
  - Para poder **acceder a los miembros de la superclase** se deberá usar el objeto `super`.
- Python
  - En cuanto a los **métodos sobreescritos**, se comporta igual que Java: cada método tiene su propia referencia.
  - Pero este lenguaje destaca por su no-privacidad, lo que se traduce principalmente en las variables.
  - **Una variable**-miembro **estará compartida por todas las clases**: cada uno **no tendrá** la suya (a diferencia de Java). **No hay ocultación**.
  - Para simular la ocultación, debe usarse `__` al inicio de la variable.
    Aunque internamente no dejará de ser una variable pública solo que empezará por el nombre de la clase
  - Para poder **acceder a los miembros de la superclase** se deberá usar la función `super()`.

Ejemplo - I

~~La subclase reconoce a los atributos de la superclase.~~ La superclase añade atributos a objetos de la subclase.

- `__dict__` es diccionario que muestra algunos atributos propios de un objeto.
- `dir(obj)` es una estructura-lista que muestra sus atributos y recursivamente las heredadas por los ancestros. Muestra `__dict__` y más atributos.

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

```
>>> class Perro(Mascota): # Subclase
...     cardinal: int = 0
...
>>> perro = Perro("Toby")
>>> print (perro.get_nombre())
Toby
```

```
>>> # Atributos únicos de las clases (1) Mascota, (2) Perro y (3) el objeto perro
>>> print([ m for m in Mascota.__dict__ if not m.startswith('__')],
...       [ m for m in Perro.__dict__ if not m.startswith('__')], perro.__dict__)
['get_nombre'] ['cardinal'] {'_nombre': 'Toby'}
>>> #
>>> # Atributos reconocidos en las clases (1) Mascota, (2) Perro y (3) 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']
```

- `Perro` no tiene constructor, así que usa `__init__()` de `Mascota`.
- En `Mascota` se asigna el valor `"Toby"` a `self._nombre`
- Con `get_nombre()` se muestra el valor de `self._nombre`
- Es decir, `self._nombre` no cambia su referencia una vez creado el atributo.

Ejemplo – II

Un ejemplo más claro donde los atributos se añaden al objeto

El atributo `self._nombre` es compartido en las dos subclases. Pero cada uno tiene su propio identificador.

```
>>> class Mascota: # Una clase
...     def __init__(self, nombre):
...         self._nombre = nombre
...     def get_nombre(self):
...         return self._nombre
...
>>> class Perro(Mascota): # Subclase
...     def __init__(self, nombre):
...     #Añade atributo nombre a self
...         super().__init__(nombre)
...     #Modifica el atributo nombre de self
...         self._nombre = "EL " + nombre
...
>>> perro = Perro("Toby")
>>> print (perro.get_nombre())
EL Toby
```

El atributo `__nombre` queda "oculto".
Realmente hay un `__nombre` en cada clase.

```
>>> class Mascota: # Una clase
...     def __init__(self, nombre):
...         # Realmente es _Mascota__nombre
...         self.__nombre = nombre
...     def get_nombre(self):
...         return self.__nombre
...
>>> class Perro(Mascota): # Subclase
...     def __init__(self, nombre):
...         super().__init__(nombre)
...         # Realmente es _Perro__nombre
...         self.__nombre = "EL " + nombre
...     def get_nombre(self):
...         return self.__nombre
...
>>> perro = Perro("Toby")
>>> print (perro.get_nombre())
EL Toby
>>> #
>>> # Esto no está bien, pero demuestra que
>>> # los atributos son siempre públicos.
>>> print (perro._Mascota__nombre)
Toby
```

Sobreescritura sobre la clase `object`

- En Python hay una clase raíz: `object`
- Algunos atributos Python conviene sobreecribirlos.
- `__init__()`. El método de inicialización de variables de un objeto.
  - Este método siempre se tienen que sobreescribir y debe de usar `super()` - salvo que sea descendiente de `object`.
- `__str__(self)`. El modo en el que un usuario vería la información de la clase. Retorna una cadena "informal".
- `__repr__(self)`. El modo en el que un programador vería la información de la clase. Retorna una cadena "formal".
- `__gt__()`, `__ge__()`, `__lt__()`, `__le__()`: Métodos que define las desigualdades lógicas al comparar dos objetos con los operadores >, >=, < y <=.
- `__eq__(self, objeto)`. Método que define el operador de igualdad (==)
  - Si `a == b` entonces `hash(a) == hash(b)`
  - Si `hash(a) == hash(b)`, entonces `a` puede ser igual a `b`
  - Si `hash(a) != hash(b)`, entonces `a != b`
  - Por tanto, se debe sobreescribir `__hash__(self, objeto)`, que es el invocado por `hash()`, para que se cumplan las condiciones indicadas.
    - https://docs.python.org/3.5/reference/datamodel.html#object.__hash__ indica cómo

Con más detalle ..

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Sobreescritura

Problema del diamante

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Problema del Diamante

Method Resolution Order(MRO)

Es lo que ocurre cuando una ancestro puede ser invocado dos veces por herencia. Python lo resuelve con `super()`
Se sigue el árbol de ancestros definido por `__mro__`

```
>>> class A:
...     def __init__(self):
...         print("A")
...
>>> class B1(A):
...     def __init__(self):
...         print("B1")
...         super().__init__()
...
>>> class B2(A):
...     def __init__(self):
...         print("B2")
...         super().__init__()
...
>>> class C(B1, B2):
...     def __init__(self):
...         print("C")
...         super().__init__()
...
>>> c = C()
C
B1
B2
A
```

Si el método no está por la rama por defecto pasa a la siguiente.

```
>>> class A:
...     def __init__(self):
...         print("A")
...
>>> class B1(A):
...     None
...
>>> class B2(A):
...     def __init__(self):
...         print("B2")
...         super().__init__()
...
>>> class C(B1, B2):
...     def __init__(self):
...         print("C")
...         super().__init__()
...
>>> c = C()
C
B2
A
```

- ¿Qué pasaría si `B2` tampoco tuviera `__init__()`?

R1: Invocaría a la clase `A` que es la siguiente de la lista.

- ¿Se puede usar un método de `A` sin pasar por las `Bx`?

R: Sí, usando `super(A, self).metodo()`.

Luis Daniel Hernández Dpto. Ingeniería de la Información y las Comunicaciones ldaniel@um.es

Tabla Resumen del Problema del Diamante

A	×	●	×	●	●	●	×	●
$B1$	×	□	●	$\bullet^{\uparrow}$	$\bullet^{\uparrow}$	$\bullet^{\uparrow}$	□	□
$B2$	×	□	×	□	●	$\bullet^{\uparrow}$	●	$\bullet\uparrow$
C	●	$\bullet^{\uparrow}$	$\bullet^{\uparrow}$	$\bullet^{\uparrow}$	$\bullet^{\uparrow}$	$\bullet^{\uparrow}$	$\bullet^{\uparrow}$	$\bullet^{\uparrow}$
	C	C A	C $B1$	C $B1$ A	C $B1$ $B2$	C $B1$ $B2$ A	C $B2$	C $B2$ A

- □ - La clase no tiene implementado el método común.
- ● - La clase tiene implementado el método pero no usa `super()`.
- $\bullet^{\uparrow}$ - La clase tiene implementado el método y también usa `super()`.
- × - Da igual lo que tenga la clase. Puede no tener el método y si lo tiene es indiferente si usa `super()` o no.

- Lo que `super()` hace es recorrer el MRO y delegar en la primera clase que encuentra por encima y que define el método que estamos buscando.

Ejemplo:

●

```
def imprime(self):
    print("Clase")
```

$\bullet\uparrow$

```
def imprime(self):
    print("Clase")
    super().imprime()
```

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Con más detalle ..

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Sobreescritura

Invocación correcta de `super()` en constructores

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Parámetros posicionales y keywords

- Consideremos:

```
>>> def fun(a, b, c, d):  # Tiene 4 parámetros
...     print(a, b, c, d)
...
```

- Si una función/método tiene *n*-parámetros podemos invocar a la función con *n*-argumentos de tal forma que el 1^{er} argumento se sustituya por el 1^{er} parámetro, el 2^{o} argumento por el 2^{o} parámetros, etc ... Son parámetros posicionales.

```
>>> fun(1, 2, 3, 4) # Invocamos con 4 posicionales.
1 2 3 4
```

- Si una función/método tiene *n*-parámetros podemos invocar a la función con *n*-argumentos indicandolo por **palabras clave**: `nombre_parametro = argumento`. Aquí las posiciones no importan, sino el nombre.

```
>>> fun(b=2, d=4, a=1, c=3) # Invocamos con 4 keywords.
1 2 3 4
```

- Se pueden usar los dos tipos a la vez; **pero** primero los posicionales y después los keywords.

```
>>> fun(1, 2, d=4, c=3) # Invocamos con 4 keywords.
1 2 3 4
```

Packing y Unpacking Argumentos

- Unpacking argumentos. Dada una función/método con varios parámetros podemos empaquetar los argumentos con *.

```
>>> def fun(a, b, c, d):
...     print(a, b, c, d)
...
>>> lista = [1, 2, 3, 4]
>>> fun(*lista)
1 2 3 4
```

- Packing argumentos. Dada una función/método podemos empaquetar varios argumentos usando un parámetro con *

```
>>> def fun(*args):
...     print(args)
...
>>> fun(1, 2, 3, 4)
(1, 2, 3, 4)
```

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Packing con Diccionarios

- El problema cuando empaquetamos argumentos es que cada uno solo se puede identificar por un índice.

```
>>> def fun(*args):
...     print(args[1]) # Muestra el segundo argumento
...
>>> fun(1, 2, 3, 4)
2
```

- Pero podemos empaquetar y usar keywords usando diccionarios, con **

```
>>> def fun(**kwargs):
...     print(kwargs) # Muestra el diccionario
...
>>> fun(a=1, b=2, c=3, d=4)
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> diccionario = {'p1': 1, 'p2': 2, 'p3':3}
>>> fun(**diccionario)
{'p1': 1, 'p2': 2, 'p3': 3}
```

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Un ejemplo con todos los tipos

- Si se quieren usar los 3 modos de pasar argumentos, el orden debe ser:
  1. posicionales,
  2. empaquetados sin keyword
  3. empaquetados con keyword

Ejemplo:

```
>>> def fun(a, b, *args, **kwargs):
...     print(a, b)    # Muestra los posicionales
...     print(args)    # Muestra los empaquetados sin keyword
...     print(kwargs) # Muestra los empaquetados con keyword
...
>>> fun(1, 2, 3, 4, 5, p1=6, p2=7, p3=8)
1 2
(3, 4, 5)
{'p1': 6, 'p2': 7, 'p3': 8}
```

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Uso del Packing en Constructores

- Se espera que cada clase hija use `super()` a la clase padre en los constructores
- Se puede usar Packing en constructores, en concreto `**kwargs`.
- La idea es que en cada `__init__()`
  - se consuman los argumentos que le son propios, y
  - pase a `super()` el resto de los argumentos.

Ejemplo:

```
>>> class A:
...     def __init__(self, pa, **kwargs):
...         print(f"En A: {pa}, Dicc:{kwargs}")
...
>>> class B(A):
...     def __init__(self, pb, **kwargs):
...         print(f"En B: {pb}, Dicc:{kwargs}")
...         super().__init__(**kwargs)
...
>>> class C(B):
...     def __init__(self, pc, **kwargs):
...         print(f"En C: {pc}, Dicc:{kwargs}")
...         super().__init__(**kwargs)
...
>>> c=C(pa='param A', pb='paramB', pc='param C')
En C: param C, Dicc:{'pa': 'param A', 'pb': 'paramB'}
En B: paramB, Dicc:{'pa': 'param A'}
En A: param A, Dicc:{}
```

Para saber más: https://rhettinger.wordpress.com/2011/05/26/super-considered-super/

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

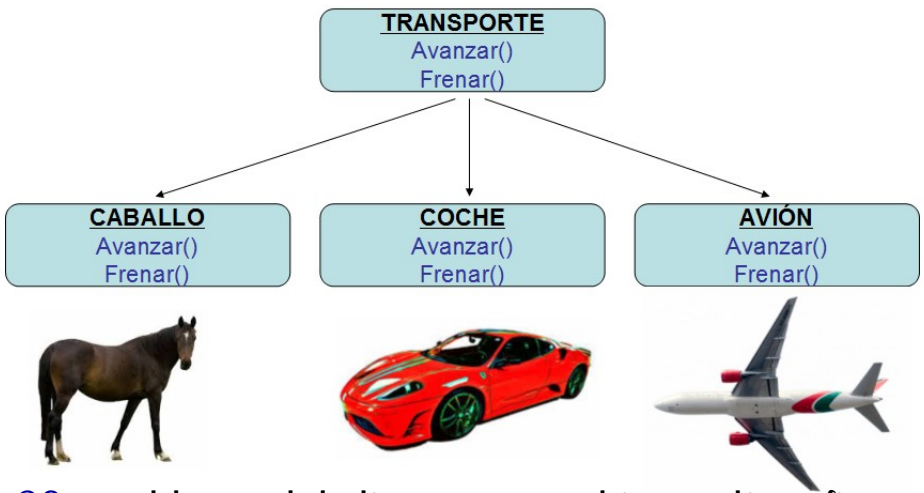
Profundizamos en ...

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Polimorfismo

- Dada una clase padre y dos clases hijas, éstas pueden heredar un método que pueden sobreescribir ya sea para reemplazarlo o para refinarlo (especialización).
- Por tanto tendremos un método que **se ejecutará de forma diferente** en cada subclase.
- El **Polimorfismo** es la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos.
  - Polimorfismo hace referencia a un conjunto de métodos con el **mismo nombre** e **igual número de Parámetros y tipos**, están **en clases diferentes** y tienen **funcionalidades diferentes**.
- El polimorfismo no necesariamente tiene que estar asociado a la herencia.

  Ejemplo: http://agrega.juntadeandalucia.es/repositorio/02122016/da/es-an_2016120212_9103251/32_polimorfismo_y_sobrecarga.html



  Ejemplo: En la pág 38, problema del diamante, se hico polimorfismo con `__init__()` en `B1` y `B2`.

Desarrollo

Lluis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Herencia

- Deberá de cumplir el **Principio de Sustitución de Liskov** (SOLID). Seminario.
- **Usa herencia** cuando
  - Representes relaciones ES-UN. Si se rompen por algún motivo... ¡Mal asunto!
  - TODA la interface de la clase A es también interface de la clase B: (1) La subclase B está siempre basada en la superclase A y (2) la implementación de la superclase A es apropiada e incluso **necesaria** para la subclase B.
  - La subclase es candidata a
    - sólo añadir nueva funcionalidad (nuevos métodos/atributos)
    - no a sobrescribir nada: dejarlo como está o añadir.
  - **Recuerda: Cuando se va a utilizar la interfaz "tal y como está".**
- **No uses herencia** cuando:
  - Si no quieres acoplamiento: afectan entre sí cuando se introducen cambios.
  - Si tu código, cuando crece por extensión, requiere modificaciones de clases que son supertipos, y además siempre suelen ser las mismas clases base las que estas tocando.
  - Cuando en las subclases tienes que reescribir métodos de la clase padre.
    - Si los Overrides empiezan a crecer y a veces de forma obligada y sin mucho sentido, smell
  - Cuando en las subclases tienes que heredar métodos que no se necesitan.
  - Cuando la superclase solo es heredada por una sola clase.
- ¿Y qué pasa si quiero objetos del mismo tipo pero no quiero acoplamiento?

¡Usa interfaces!

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Agregación/Composición

- Representar relaciones TIENE-UN, PARTE-DE (, USA-UN)
- Debe usarse cuando **parte de la interface** del tipoA también lo son para el tipoB
  Recuerda: Cuando se va a reutilizar sin mantener la interfaz.
- Se adapta mejor a los cambios, es más flexible, que la herencia.
  - Modela una relación débilmente acoplada.
  - Los cambios en una clase de componente tienen un efecto mínimo o nulo en la clase compuesta.
  - Los diseños basados en la composición son más adecuados para cambiar.
- ¿Y qué pasa si quiero objetos del mismo tipo pero no quiero acoplamiento?
  ¡Usa interfaces!
- ¿Y qué pasa si quiero objetos con polimorfisimo pero no quiero acoplamiento?
  ¡Usa interfaces!
- Se podría hablar más: aprende **patrones de diseño**.

Observaciones

- Retoma esta sección cuando estudies las interfaces.
- Ejemplo comparativo: https://realpython.com/inheritance-composition-python/

Ejercicio

Ejemplo basado de https://youtu.be/NcVgYxHfCrs?t=1457

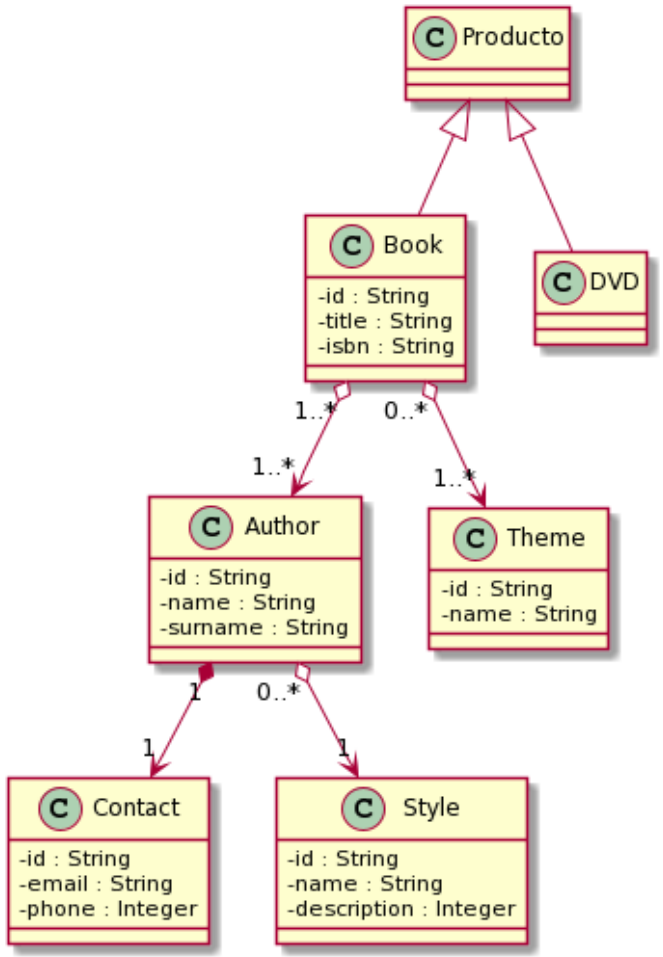
Los libros y los DVDs son productos culturales. En concreto, los libros tienen un título, ISBN y una lista de autores. Los libros están asociados a un conjunto de temas. Un tema puede estar asociado a muchos libros. De cada autor queremos guardar su nombre y su apellido y sus datos de contacto, que están formados por un email y un teléfono. Un autor tiene un estilo, pero un estilo puede ser utilizado por muchos autores. De un estilo se guarda el nombre y la descripción.
Haz la representación UML de esta situación.

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Ejercicio

Ejemplo basado de https://youtu.be/NcVgYxHfCrs?t=1457

Los libros y los DVDs son productos culturales. En concreto, los libros tienen un título, ISBN y una lista de autores. Los libros están asociados a un conjunto de temas. Un tema puede estar asociado a muchos libros. De cada autor queremos guardar su nombre y su apellido y sus datos de contacto, que están formados por un email y un teléfono. Un autor tiene un estilo, pero un estilo puede ser utilizado por muchos autores. De un estilo se guarda el nombre y la descripción.
Haz la representación UML de esta situación.



Una posible solución. Notar cómo se representan las dos jeraquías.

Ejercicio

Ejercicio.

En un sistema de simulación están los **agentes** (inteligentes) con métodos que permiten realizar una toma de **decisiones** (p.e. con una máquina de estados) y aquellos que además tienen un **cuerpo** físico y que les permite moverse en el escenario.

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.

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 desactivas. Pero si es una alarma **luminosa**, encenderá una luz. En el caso de que sea sonora y luminosa hará las dos cosas.

Ejercicio

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?

Ejercicio.

Están los relojes (con sus horas, minutos y segundo), que cada. vez que hacen un tictac avanzan un segundo. También están los calendarios (con sus días, meses y años) que avanzan día a día. Y están los relojes con calendarios. ¿Cómo se construyen? ¿cómo avanzan un segundo? ¿cómo avanzan un día?