Paradigma
Orientado a Objetos

Módulo 13:
Herencia.
Super.
Redefinición.

por Fernando Dodino
Versión 2.2
Septiembre 2018

Indice

1 Herencia

1.1 Enunciado base

Un ornitólogo, luego de estudiar el comportamiento de las golondrinas tijerita como pepita, se dio cuenta de que existen otros 2 tipos de aves que le interesan:

- el petrel
- y la torcaza

Todas saben volar y comer igual que pepita, pero

- al ornitólogo le interesa saber la cantidad de kilómetros que vuela un petrel
- la torcaza es medio atolondrada, antes de comer se pone tan contenta que vuela en círculos 1 kilómetro

1.2 Intro a superclase

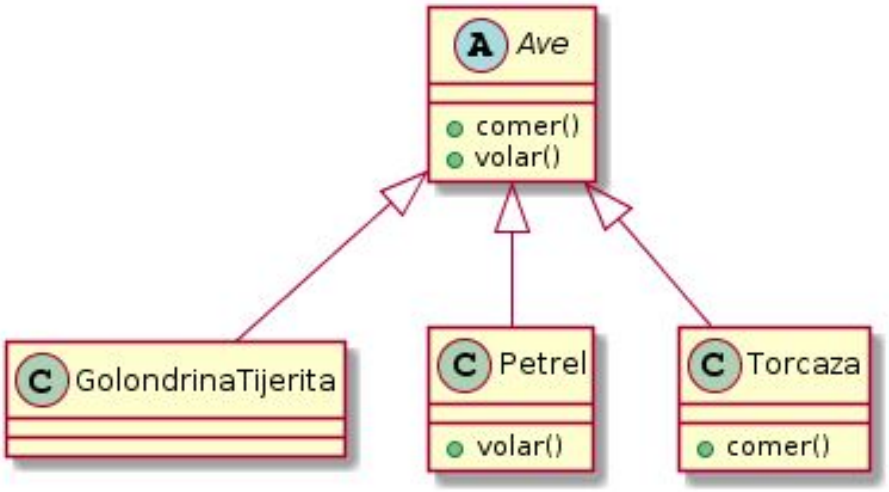
La golondrina tijerita, el petrel y la torcaza saben comer y volar. Esto las hace en principio **polimórficas**, pero además la forma en que comen y vuelan es muy similar, aunque no exactamente igual.

Sabemos que una clase es el concepto que permite agrupar el comportamiento (y la estructura) de cada una de las instancias. Pero como el petrel “vuela distinto” (para nuestro sistema) y la torcaza “come distinto”, esto nos lleva a modelar tres clases diferentes: GolondrinaTijerita, Petrel y Torcaza. Ahora bien, no queremos repetir el mismo código para las 3 clases al comer y al volar.

Afortunadamente, podemos relacionar una clase con otra: GolondrinaTijerita, Petrel y Torcaza pueden tener a Ave como clase madre o **superclase**. Las superclases tienden a representar un concepto más general, mientras que las subclases se enfocan a especializaciones o casos particulares.

1.3 Representación

Generamos la representación en el diagrama de clases:



La flecha con el triángulo cerrado marca la relación de herencia. Las subclases de Ave heredan a priori sus atributos y comportamiento.

2 Herencia en Wollok: ejemplo de las aves

Recordemos cómo es el comportamiento de una golondrina tijerita

- cuando vuela, consume un joule para cada kilómetro que vuela, más 10 joules de "costo fijo" en cada vuelo
- cuando come, adquiere 4 joules por cada gramo que come

Este comportamiento puede ahora ser la definición por defecto de un Ave. Creamos la clase Ave en Wollok[1]:

```
/**
* Definición por defecto de un Ave
*/
class Ave {
	var energia = 50

	method volar(kilometros) {
		energia = energia - kilometros + 10
	}

	method comer(gramos) {
		energia = energia + gramos * 4
	}
}
```

[1] El ejemplo completo puede descargarse de https://github.com/wollok/herencia-aves-pepita

2.1 Golondrina Tijerita

¿Qué sucede con la golondrina tijerita? Se comporta ahora “igual que un ave”. Esto lo definimos así:

```
class GolondrinaTijerita inherits Ave { }
```

Entonces una golondrina tijerita, por heredar de Ave

- tiene energía
- sabe comer y volar, como lo hacen todas las aves

Además agrega un comportamiento que no viene al caso en este momento:

```
class GolondrinaTijerita inherits Ave {
     method hacerAlgo(conCosa) { ... }
}
```

Podemos verlo al instanciar una golondrina tijerita en la consola REPL:

```
>>> const pepita = new GolondrinaTijerita()
a GolondrinaTijerita[energia=50]
>>> pepita.volar(10)
>>> pepita
a GolondrinaTijerita[energia=30]
```

Al instanciar a pepita, tenemos una referencia llamada energía, y podemos enviarle el mensaje volar. Pero ¿cómo es que pepita entiende volar?

2.2 Method lookup de pepita

Tenemos que revisar la definición del *method lookup*. Hasta ahora, al enviar un mensaje a un objeto,

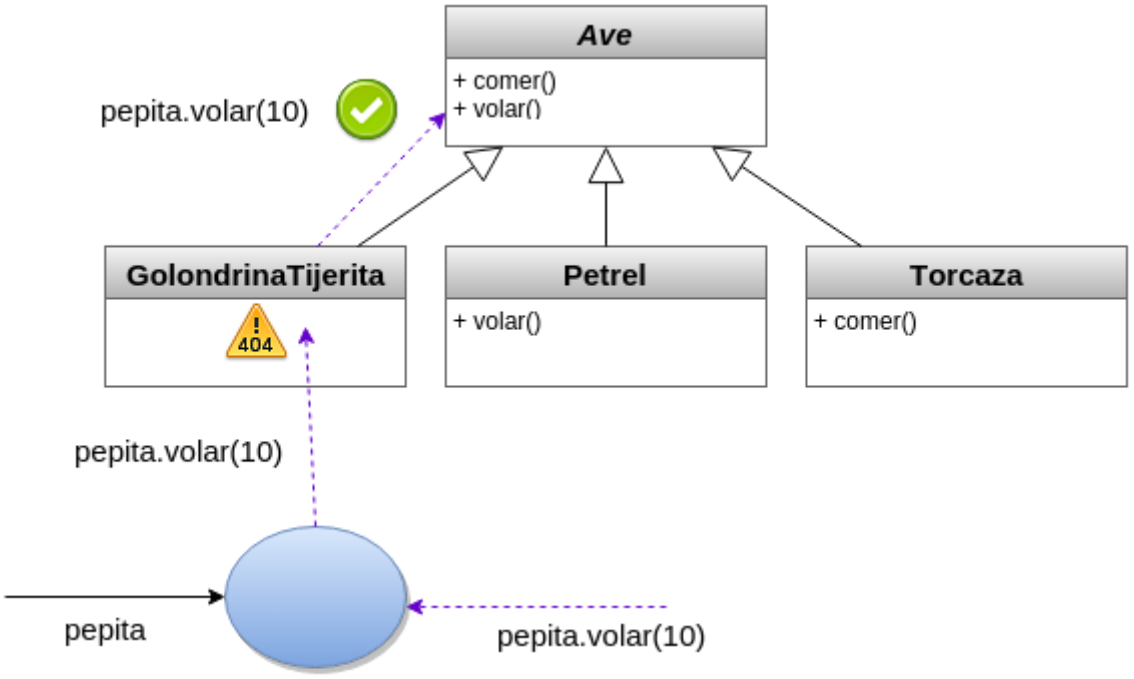
- si se trata de un objeto autodefinido o “well-known object”, se ejecuta el método que está en dicho objeto
- si es instancia de una clase, comenzamos buscando el método en la clase a la que pertenece dicha instancia. En el ejemplo de pepita, buscamos el método “volar(kilometros)” en la clase GolondrinaTijerita.

Ahora extendemos lo que sucede si no encontramos la definición en la clase del objeto receptor:

- la búsqueda continúa en la superclase, en este caso Ave. Aquí tenemos definido un método volar con un parámetro y se evalúa para los valores que tiene pepita.

- En el caso de no existir en la superclase, se sigue buscando en la clase superior, hasta llegar a Object (la clase madre de todas las clases). Y finalmente si no encontramos allí el método que buscamos, recibimos un mensaje de error.

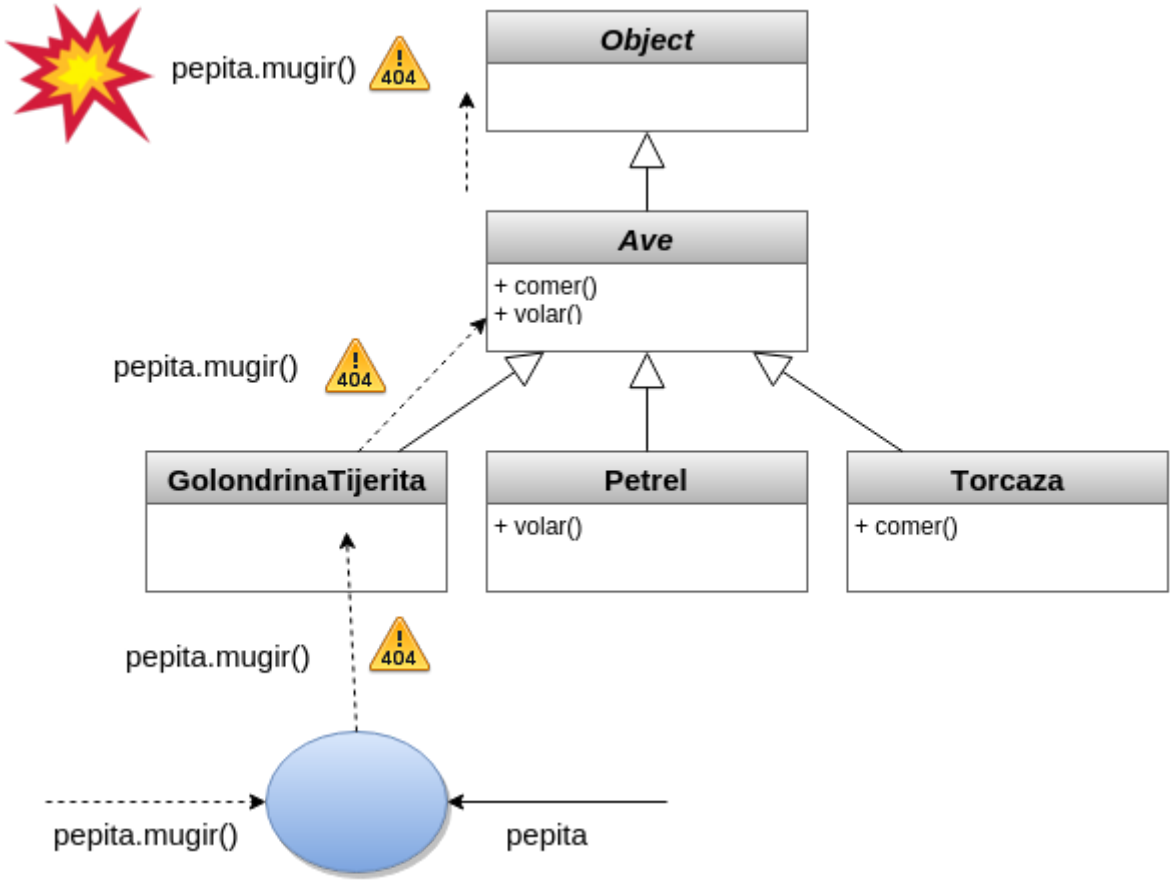
Repasamos el gráfico en el envío del mensaje pepita.volar(10):



Vemos qué pasa cuando enviamos un mensaje que no puede ser respondido:

```
>>> pepita.mugir()
wollok.lang.MessageNotUnderstoodException: un/a
GolondrinaTijerita[energia=50] no entiende el mensaje mugir()
   at
wollok.lang.Object.messageNotUnderstood(messageName,parameters)
(classpath:/wollok/lang.wlk:228)
```

Y lo vemos gráficamente:



3 Segundo caso: el Petrel

El petrel

- come y vuela igual que un ave
- además el ornitólogo quiere registrar la cantidad de kilómetros que voló

La primera pregunta que podríamos hacernos es: ¿es responsabilidad del petrel registrar los kilómetros de vuelo? ¿no es el ornitólogo el que termina haciendo el trabajo?

3.1 Modelo y realidad

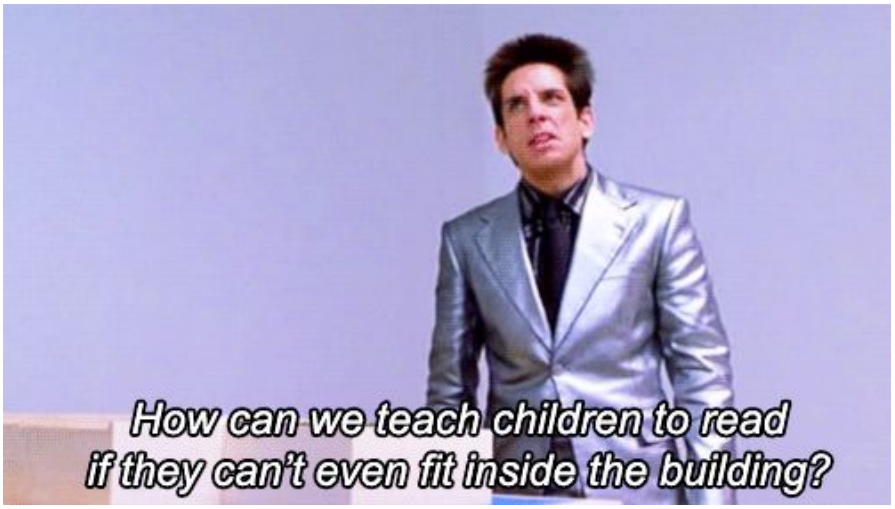
Claro, pero ¡¡ojo!! Acá nos estamos confundiendo modelo y realidad

- en la realidad, el petrel vuela, el ornitólogo anota en un cuaderno las veces que voló
- en nuestro modelo, el petrel no es el petrel, sino una representación del petrel que es mucho menos compleja, a la que le sacamos todas las características que no son esenciales mediante el proceso de abstracción.

Dejamos al lector dos ideas asociadas:
El cuadro de René Magritte, “Esto no es una pipa”

En él, Magritte resalta la idea de que dibujar una pipa no constituye una pipa en sí, sino una representación de la pipa (acotada por el observador).

En la película Zoolander[2], al protagonista Ben Stiller le muestran una maqueta de un "Centro para niños que no saben leer bien":



[2] Idea extraída de una clase de nuestro amigo Pablo Beláustegui

Indignado, el personaje Derek Zoolander replica: *"¿Qué es esto, un centro para hormigas? ¿Cómo vamos a enseñarle a los chicos a leer si es imposible que quepan en el edificio?"*

Claro, Zoolander también confunde modelo y realidad.

3.2 Volviendo al código de petrel

Entonces el petrel de nuestro ejemplo se puede implementar así:

```
class Petrel inherits Ave {
	var kilometrosVolados = 0

	override method volar(kilometros) {
		super(kilometros)
		kilometrosVolados = kilometrosVolados + kilometros
	}
}
```

Antes de explicar lo que hicimos arriba, lo probamos:

```
>>> const unPetrel = new Petrel()
>>> unPetrel.volar(10)
>>> unPetrel.volar(20)
>>> unPetrel
a Petrel[energia=0, kilometrosVolados=30]
```

Un detalle: además de los kilómetros volados (definidos en Petrel), el petrel tiene energía, porque dijimos que un Petrel "es un" Ave.

3.3 Redefinición de métodos

¿Por qué aparece la palabra override antes de method?
Porque la subclase está **redefiniendo** comportamiento de la superclase: ya teníamos una definición de volar(kilometros) pero nosotros queremos escribir otra, que va a pisar a la definición original de Ave.

Entonces al enviar el mensaje

```
>>> unPetrel.volar(10)
```

el method lookup comenzará a buscar en la clase receptora de la instancia unPetrel. Esto es... la clase Petrel, que ahora sí tiene una definición de volar.

3.4 Super

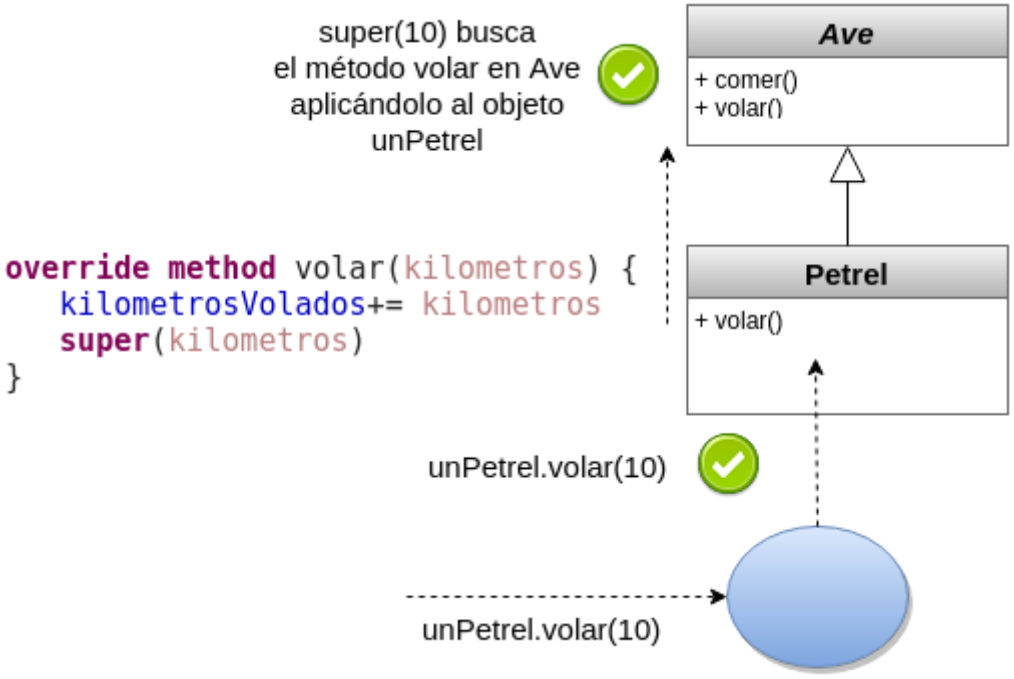
El problema es que yo quiero aprovechar el comportamiento de volar() que está definido en Ave, pero sin repetir el código. Por eso utilizamos la pseudo-variable super:

```
override method volar(kilometros) {
	super(kilometros)
	kilometrosVolados = kilometrosVolados + kilometros
}
```

Esto permite alterar el method lookup en el contexto en donde estamos, salteando la clase del objeto receptor y comenzando por su superclase. Esto es útil particularmente cuando estamos en un método redefinido y queremos evitar un loop infinito de llamadas al mismo método en el cual estamos.

3.5 Method lookup de unPetrel

Mostramos gráficamente cómo es el method lookup para el mensaje `unPetrel.volar(10)`:



4 Tercer ejemplo: la torcaza

La torcaza

- vuela como un ave

- y al ser medio atolondrada, por un lado come igual que un ave pero antes de comer se pone tan contenta que vuela en círculos 1 kilómetro

Ya sabemos que si heredamos de Ave nuestra definición de volar() no debe cambiar, debemos redefinir el método comer().

4.1 Self

Al comer, tenemos que hacer que vuele un kilómetro. Pero no queremos repetir el código de volar() dentro del método comer():

```
class Torcaza inherits Ave {

	override method comer(gramos) {
		energia = energia - (1 + 10) // copio el método volar, je
		super(gramos)
	}
}
```

Lo que podríamos hacer es enviar el mensaje volar(1), pero ¿a quién? Al mismo objeto receptor, usamos para eso la referencia *self*:

```
class Torcaza inherits Ave {

	override method comer(gramos) {
		self.volar(1)
		super(gramos)
	}

}
```

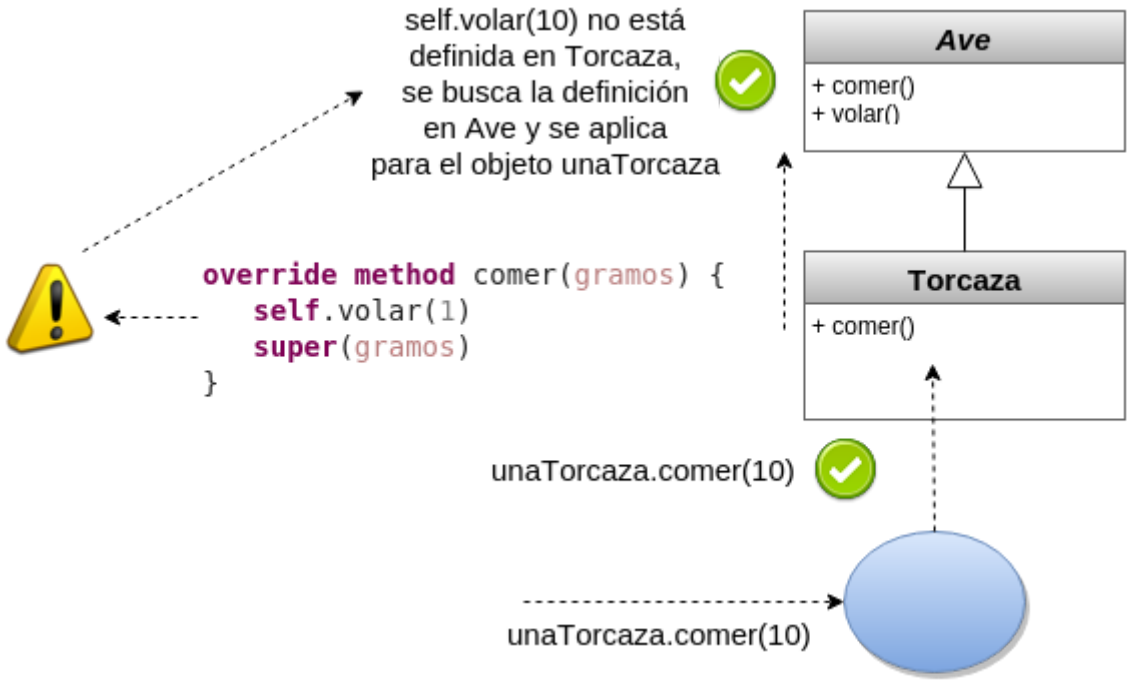
En general

- usamos *self* para enviar mensajes al objeto receptor
- y cuando no podemos usar *self*, porque entraríamos en loop infinito, usamos *super*

La diferencia está en que *self* no cambia el method lookup, que comienza por la clase del objeto receptor, mientras que *super* saltea el primer paso y comienza la búsqueda del método en la superclase de donde se invoca *super*. Pero el objeto receptor **nunca cambia, no hay otro objeto más que la torcaza**.

4.2 Method lookup de torcaza

Vemos el gráfico que muestra el lookup de la primera línea del método comer para torcaza:



El method lookup de la segunda línea es similar al que vimos en petrel:

- super(gramos) busca la definición del método comer(gramos) en Ave y la aplica para la torcaza

5 Más sobre herencia

5.1 Clases abstractas y concretas

Ave es una **clase abstracta**: tiene sentido como una forma de agrupar comportamiento y atributos para las subclases, no para generar instancias de Ave.

- En algunos lenguajes esta definición es explícita: no podés crear una instancia de Ave si dijiste que era una clase abstracta. Al hacer new Ave() el compilador chequea que Ave no sea abstracta y en ese caso se produce un error.

```
public abstract class Ave { ... // Definición explícita en Java
```

- En Wollok, esta definición ocurre naturalmente: si no hago new Ave(), esto implica que Ave termina siendo una clase abstracta, no hace falta decirlo.

Una pregunta frecuente que surge es: ¿las superclases deben ser abstractas? ¿puede haber superclases concretas? La respuesta es: las clases son concretas si tiene sentido instanciarlas, sean superclases, subclases o clases “sueltas” (que sólo hereden de *Object*). Y si una clase es abstracta, seguramente es porque tiene subclases que redefinen algún comportamiento. De otra manera, ¿para qué

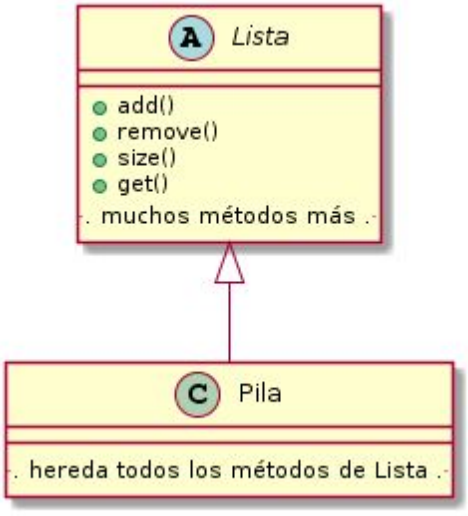
querríamos generar una clase abstracta sin subclasificarla luego? No podemos instanciarla y no sirve como agrupador de ninguna jerarquía, es difícil encontrarle sentido.

5.2 Herencia vs. composición

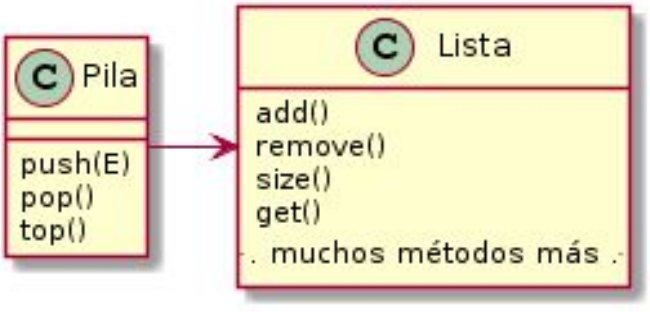
La herencia marca una relación entre clases (es estática), la superclase tiene características más generales mientras que la subclase toma comportamiento específico y cuando es necesario lo redefine. **En la composición** no hay una jerarquía de clases, sino **que intervienen dos instancias**: una conoce a la otra y le envía mensajes.

En los lenguajes donde tenemos herencia simple, la herencia es un mecanismo más limitado que la composición: la taxonomía de las clases tiene un único punto de vista. El ejemplo “de libro” es que al modelar una clase Perro, tengo que pensar si quiero que la jerarquía esté basada en animales domésticos y salvajes, o en vertebrados e invertebrados, o en mamíferos, ovíparos, ovovivíparos, etc. No puedo tener más de un punto de vista, y esto trae complicaciones por ejemplo al representar un String: ¿debería heredar de una clase Collection (porque es una colección ordenada de caracteres) o de una clase que sepa decirnos si es mayor o menor que otro String, algo así como un Ord de Haskell?

¿Y por qué comparar a la herencia con la composición? Supongamos que tenemos que modelar una Pila. La pregunta que nos tenemos que hacer es: ¿una pila *es una* lista? ¿o una pila *tiene* una lista?



Pila es una Lista



Pila tiene una Lista

Si Pila hereda de Lista, la ventaja es que toma todo el comportamiento de su superclase, y también allí radica su principal desventaja: quizás no sea necesario tener una interfaz tan grande, podríamos tener sólo tres métodos para la pila:

- push(): pone un elemento de la pila
- top(): muestra el elemento que está arriba de todo en la pila
- pop(): saca el elemento que está arriba de todo en la pila

Entonces lo que nos conviene es que una Pila tenga una Lista, pero su implementación quede encapsulada en la Pila. El que utiliza a la Pila no necesita saber cómo está construida, simplemente utiliza los tres mensajes que definimos para la Pila.

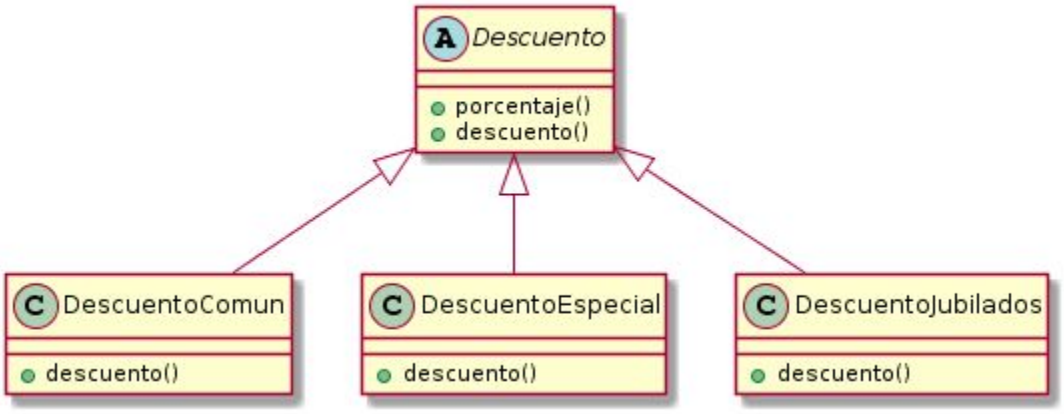
Mediante la composición, permitimos que a futuro la Pila herede de otra superclase que tenga más cosas en común que una Lista, sin gastar el *único tiro* que la herencia provee.

5.3 ¿Instancias o clases?

Dado el siguiente requerimiento

"A la hora de vender, tenemos tres tipos de descuento. El descuento común es del 5%, el especial de un 10% y para jubilados es un 20%".

A primera vista podríamos pensar en subclasificar Descuento en: DescuentoComun, DescuentoEspecial y DescuentoJubilados.



En la clase abstracta Descuento el método porcentaje se escribiría

```
>>Descuento
method porcentaje() = self.descuento() / 100
method descuento()
```

El método descuento no tiene cuerpo, es un **método abstracto** (solo sirve para forzar a que las subclases implementen dicha interfaz), y provoca entonces que la clase Descuento sea abstracta.

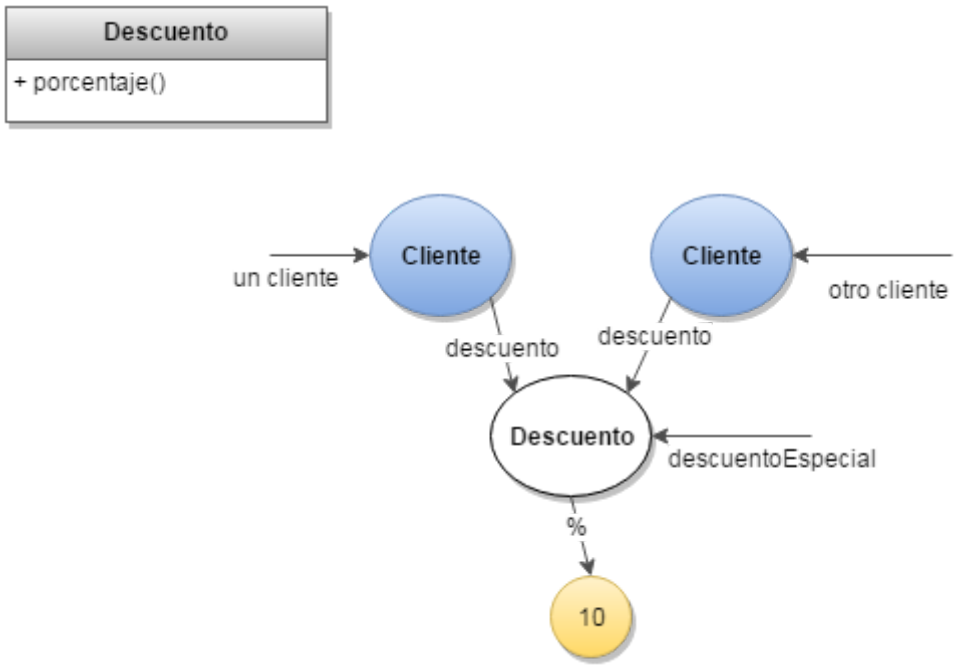
Pero ¿cómo implementaríamos el método descuento en cada subclase?

```
>>DescuentoComun
override method descuento() = 5

>>DescuentoEspecial
override method descuento() = 10

>>DescuentoJubilados
override method descuento() = 20
```

En realidad estamos repitiendo la misma idea, ya que lo único que difiere es el % de descuento. En ese caso podemos modelar una clase Descuento con tres instancias diferentes, una para cada tipo de descuento.



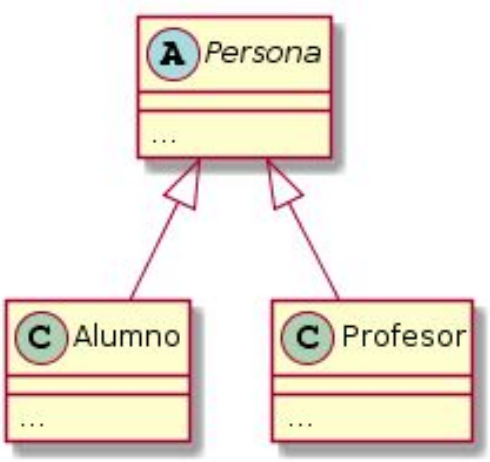
descuentoEspecial es una referencia a una instancia de Descuento

Otro ejemplo: “Modelar las piezas de ajedrez.” ¿tengo una clase Caballo, otra Alfil, otra Rey o me arreglo con 32 instancias de Pieza?

Al modelar con objetos: ¿cuándo usar instancias y cuándo usar clases? Uso clases cuando **hay comportamiento diferente (cuando el código es diferente)**.

5.4 Cambiar la clase

Si tenemos esta jerarquía de clases



Todo parece funcionar bien: el alumno y el profesor pueden en algún contexto ser polimórficos pero cada uno define su propio comportamiento. Ahora, cuando un buen alumno finaliza su cursada, le ofrecen ser ayudante y pasa a ser profesor, entonces tenemos un problema: una instancia no puede pertenecer a dos clases, y tampoco es fácil cambiar una referencia a un alumno por el de un profesor.

De la misma manera jerarquías como casados / solteros, felices / aburridos / indignados, sin hijos / con hijos están basadas en características *temporales* de un objeto y no deberían elegirse como criterio de subclasificación. Por eso una buena práctica para elegir el punto de vista para armar la jerarquía de clases es buscar aquello que sea intrínseco al objeto, de forma tal que un **objeto pertenezca exactamente a una clase durante todo su ciclo de vida**.

6 Herencia de WKO

Volviendo al objeto pepita original:

```
object pepita {
	var energia = 100

	method energia() = energia
	method volar(kms) { energia = energia - (kms + 10) }
	method comer(gramos) { energia = energia + (4 * gramos) }
}
```

Incorporamos otra ave chichita, que

- vuela igual que pepita
- pero come distinto

Pero además, nuestro usuario nos avisa que hay infinidad de otras aves que vuelan y comen de la misma manera que pepita. Tanto *chichita* como *pepita* son dos objetos importantes en nuestra solución, no queremos perder la posibilidad de referenciarlas y mandarles mensajes. Pero no pasa eso con las otras aves: para ellas se pierde el sentido de la individualidad, entonces no necesito generar un *named object* para cada uno, me basta con que sean instancias de una clase Ave.

¿De qué manera podemos reutilizar el comportamiento de pepita, chichita y las demás aves? Wollok permite que un WKO (un objeto conocido) herede de cualquier clase. Entonces podemos mudar el comportamiento de pepita a una clase Ave, y definir el comportamiento específico para chichita:

```
class Ave {
	var energia = 100

	method energia() = energia
	method volar(kms) { energia = energia - (kms + 10) }
	method comer(gramos) { energia = energia + (4 * gramos) }
}

object pepita inherits Ave { ... }

object chichita inherits Ave {
	override method comer(gramos) {
		energia = energia * gramos
	}
}
```

Fíjense que:

- pepita toma el comportamiento de Ave (y debería agregar algún comportamiento individual)
- chichita también toma la definición de la clase Ave pero además pisa el método comer. Como hereda de Ave tiene acceso a su estructura interna, en este caso la referencia `energia` se puede utilizar directamente.

¿Cómo vuela pepita? Igual que un ave
¿Cómo vuela chichita? Igual que un ave
¿Cómo come chichita? Como solo ella lo sabe hacer

6.1 Method lookup de un WKO con herencia

El method lookup de un objeto se resuelve de la siguiente manera: al enviar un mensaje a un WKO

1. primero se busca la definición en el objeto

2. si no se encuentra un método y el objeto hereda de una clase, se busca en esta clase
3. ¿y si no está en dicha clase? Se busca en la superclase hasta encontrarla o bien llegar a Object y fallar.

6.2 Objetos y clases polimórficos

Con el ejemplo anterior, el lector habrá notado que tenemos un set de objetos polimórficos:

- pepita
- chichita
- y las instancias de la clase Ave

Todos entienden los mensajes energia(), volar(kms) y comer(gramos).

En un REPL podemos enviar mensajes sin tener que preocuparnos si son WKO o instancias de una clase

```
>>> pepita.volar(10)
>>> chichita.volar(5)
>>> const tweety = new Ave()
a Ave[energia=100]
>>> tweety.volar(6)
>>> [pepita, new Ave(), tweety, chichita].forEach ({ ave =>
ave.volar(5) })
```

7 Resumen

Las clases pueden organizarse jerárquicamente de la más general a la más particular: el mecanismo de herencia permite reutilizar comportamiento y atributos para evitar duplicación de código, y redefinir en cada subclase lo que sea necesario en cada caso. El method lookup sigue la búsqueda desde la clase del objeto receptor por todas las superclases, hasta encontrar el método o fallar. Mientras que

self permite enviar un mensaje al propio objeto, *super* cambia el mecanismo de method lookup salteando la clase donde está definido el método y comenzando directamente en la superclase.

Por último, a la hora de modelar con objetos es conveniente tener en claro que para elegir la subclasificación debe haber comportamiento diferencial entre las subclases, y tener en cuenta el contrapunto entre herencia y composición como alternativas, cada una con sus puntos a favor y en contra.