Paradigma
Orientado a Objetos

Módulo 08: Clases.
Method lookup y polimorfismo con clases.

por Valeria Pennella
Fernando Dodino
revisado por Matías Freyre
revisado por Lucas Spigariol
Versión 1.4
Septiembre 2019

Indice

1 Introducción al concepto de clase

Necesitamos modelar un sistema para un comercio de ventas de libros. Sabemos que vendemos libros como:

- El Principito
- Martín Fierro
- El Reino del Revés

Vamos a tener 3 objetos, donde cada uno representa un libro diferente pero todos tienen características similares. Cuando a cada libro le pregunte si es caro cada uno va a responder lo que corresponda según su precio. Es decir

```
>>> elPrincipito.esCaro()
>>> martinFierro.esCaro()
>>> elReinoDelReves.esCaro()
```

Si creamos El Principito tendríamos esta definición:

```
object elPrincipito {
	var property precio = 100
	method esCaro() = precio > 150
	method aumentar(aumento) {  precio = precio + aumento}
}
```

Considerando que el precio inicial de cada libro es 100, pero luego cada uno puede variar independientemente definimos los otros dos libros de la siguiente manera:

```
object martinFierro {
	var property precio = 100
	method esCaro() = precio > 150
	method aumentar(aumento) {  precio = precio + aumento}
}

object elReinoDelReves {
	var property precio = 100
	method esCaro() = precio > 150
	method aumentar(aumento) {  precio = precio + aumento}

}
```

Luego, podemos hacer

```
>>> elPrincipito.aumentar(80)
>>> martinFierro.aumentar(50)
>>> elReinoDelReves.aumentar(100)
```

y ante la misma pregunta, las respuestas van a ser diferentes

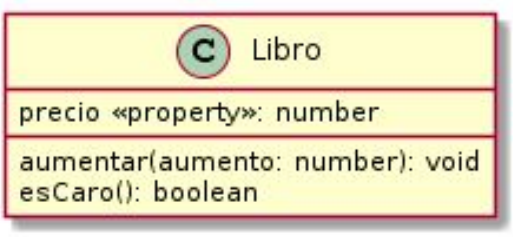
```
>>> elPrincipito.esCaro()     // true
>>> martinFierro.esCaro()     // false
>>> elReinoDelReves.esCaro()  // true
```

Lo que vemos es que si bien cada uno de los libros tienen un precio diferente, todos los libros tienen el mismo comportamiento para decir si es caro; también es la misma la forma de aumentar el precio. Para no tener código duplicado podemos definir una clase Libro donde esté definido el comportamiento de todo libro. Es decir...

```
/**
 * Definición de la clase Libro
 */
class Libro {
	var property precio = 100
	method esCaro() = precio > 150
	method aumentar(aumento) { precio = precio + aumento }
}
```

De esta manera, toda la lógica que teníamos definida en cada uno de los objetos, como es igual para todos, podemos tenerla directamente en un solo lugar que es la Clase.

Representamos la clase Libro en un diagrama de clases UML



Aquí vemos que una clase se representa con un rectángulo con 3 divisiones:

- arriba está el nombre de la clase
- en el medio las referencias, ya sean constantes o variables
- en el último compartimento escribimos los métodos

2 Clase e instancia

¿Y cómo hago ahora para crear cada uno de los libros? A partir de la clase Libro, voy a crear cada uno de los diferentes ejemplos concretos.

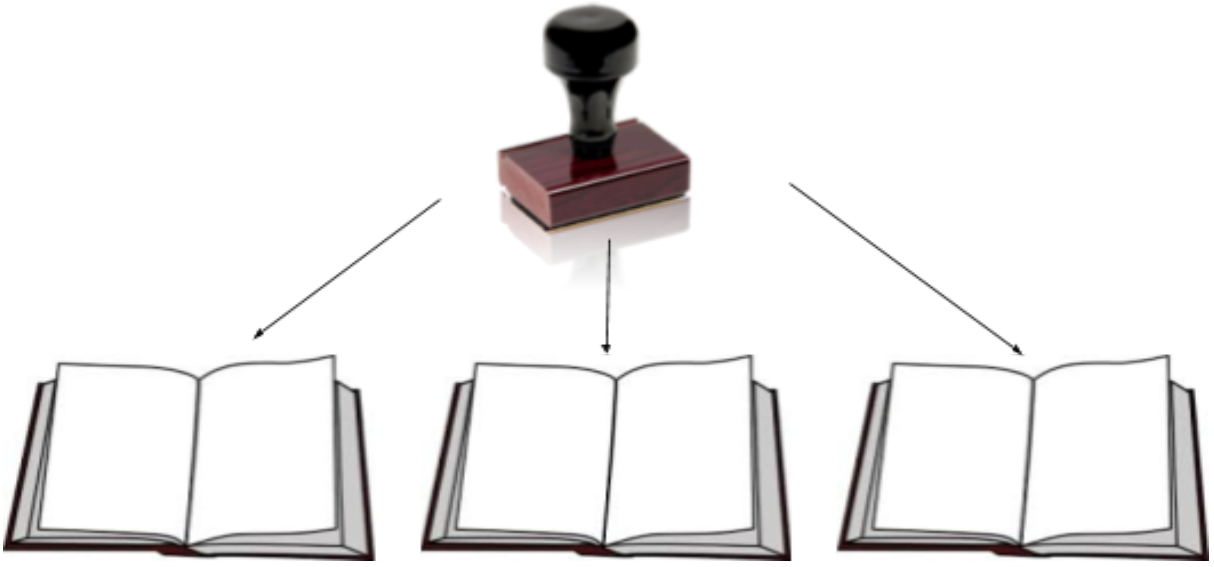


Fig.1: La clase puede verse como un *sello* con el que generamos las instancias

Otra analogía es ver la clase Libro como un molde a partir del cual vamos a crear las instancias es decir:

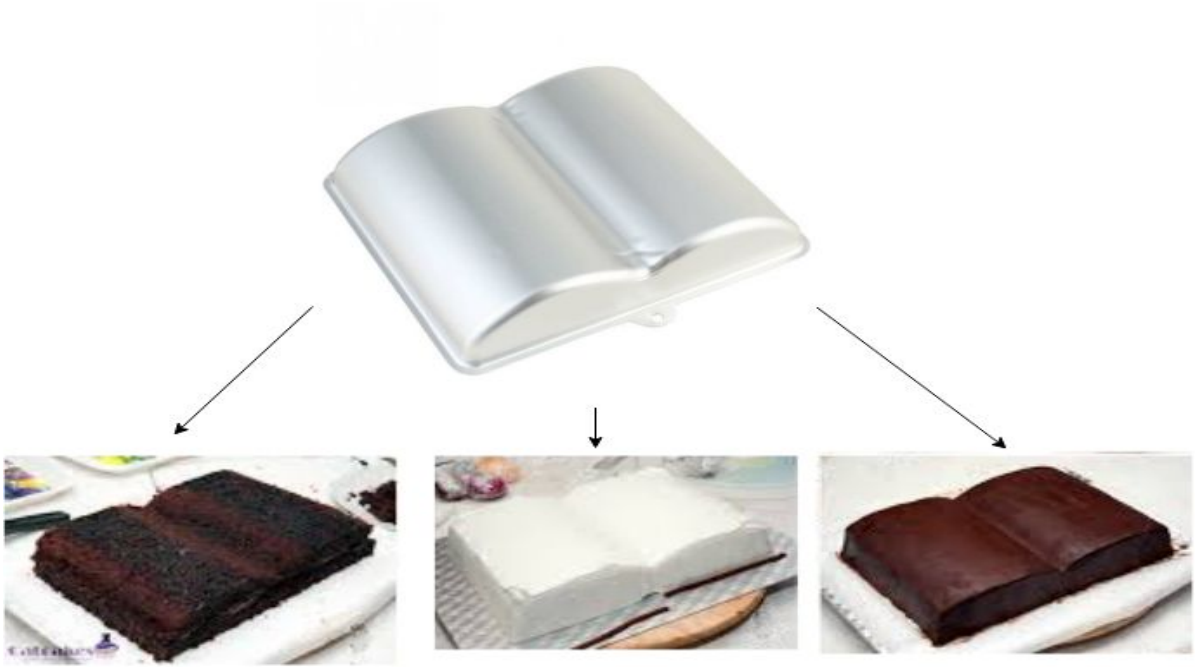


Fig. 2: con el *molde* (hueco) instanciamos una chocotorta, una torta glaseada y una bombón

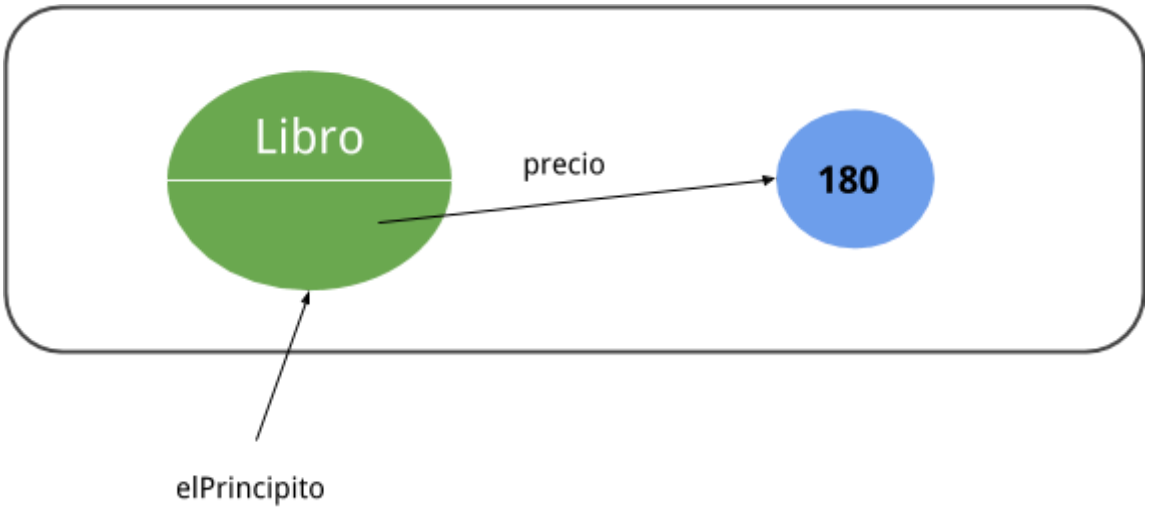
Para crear el objeto `elPrincipito` escribimos:

```
>>> const elPrincipito = new Libro()
```

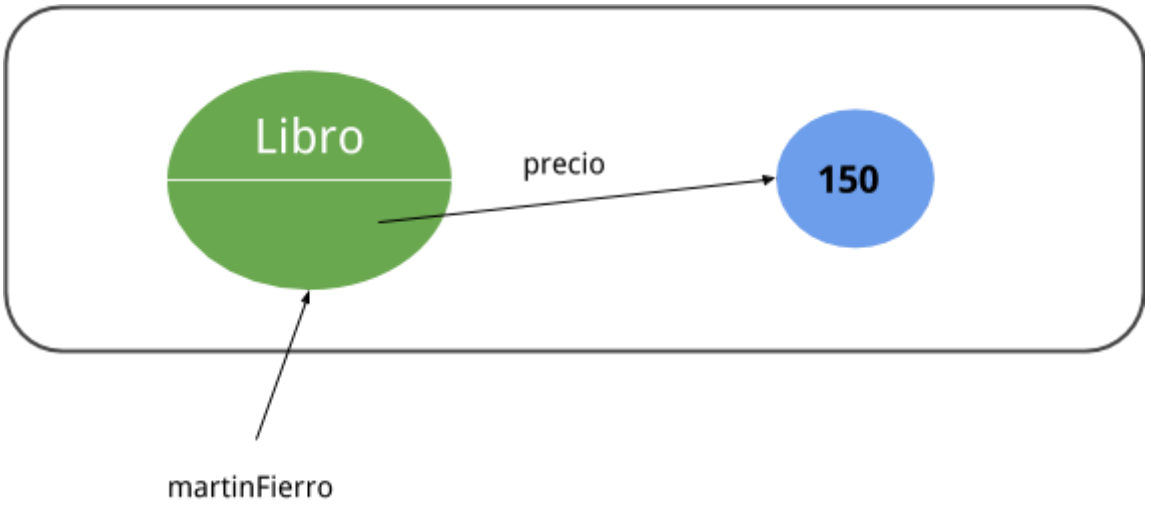
Para tener un nuevo libro creamos con new, una nueva instancia[1] de la clase Libro. Por lo tanto la variable `elPrincipito` va a apuntar a una instancia de `Libro`.

Y voy a poder decirle al principito que aumente de precio y preguntarle si es caro

```
>>> elPrincipito.aumentar(80)
>>> elPrincipito.esCaro()
```



De la misma manera puedo crear una referencia a martinFierro, que será un libro.



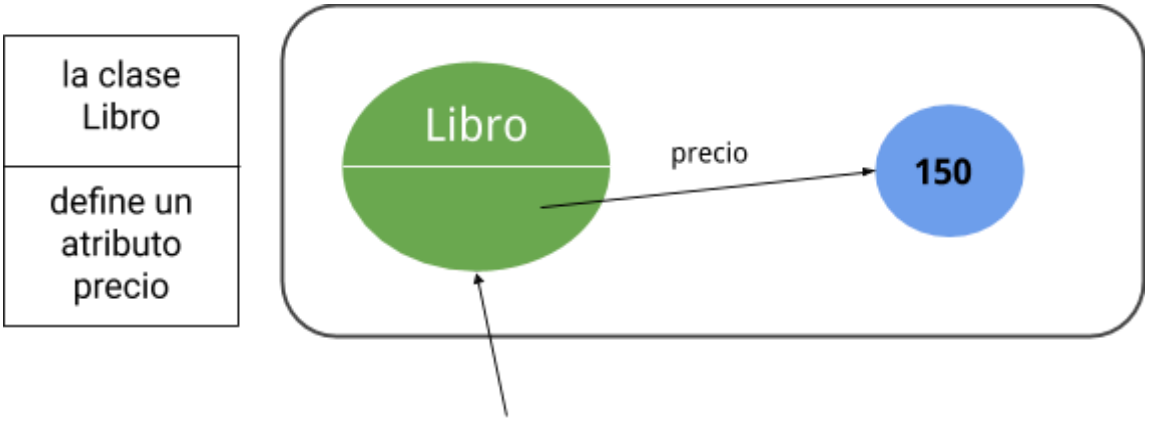
[1] Hablamos de instancia como sinónimo de objeto

3 Estructura de un objeto

Al instanciar un objeto, éste tiene todas las referencias definidas en su clase (lo que también se llama **estructura**), solo que con sus valores particulares. En el ejemplo anterior, Libro define que todo libro tiene una referencia a precio.

El valor de cada libro es propio de cada instancia, por eso las referencias o atributos se llaman también **variables de instancia**.

```
>>> const martinFierro = new Libro()
>>> martinFierro.aumentar(50)
```



martinFierro, es un objeto que toma la referencia "precio" de la clase Libro, apuntando al valor concreto 150

4 Method lookup con clases

Nuestros objetos ya no tienen comportamiento, sino que definimos esa responsabilidad en la clase. Debemos entonces cambiar nuestro mecanismo de method lookup.

Según esta definición: qué sucede cuando enviamos el mensaje...

```
>>> elPrincipito.esCaro()
```

elPrincipito no es un objeto que tenga comportamiento explícito. Es una instancia de una clase Wollok. Entonces vamos a buscar la definición a la clase a la que pertenece elPrincipito. Ahí vemos que existe una definición del método esCaro

```
method esCaro() = precio > 150
```

Y se termina resolviendo para los atributos propios del objeto elPrincipito: como la referencia precio de este objeto apunta a 180, elPrincipito es un libro caro.

De la misma manera si preguntamos si el Martín Fierro es caro...

```
>>> martinFierro.esCaro()
```

Nuevamente vamos a buscar la definición a la clase a la que pertenece martinFierro, que es como suponemos la clase Libro.

```
method esCaro() = precio > 150
```

Y se termina resolviendo para los atributos propios del objeto martinFierro: como precio referencia a 150, martinFierro **no** es un libro caro.

5 Clases y tipos

Un tipo es un conjunto de mensajes que entiende un objeto.

Un caso particular es considerar todos los mensajes que entiende un objeto, entonces el tipo está definido por la clase. En nuestro ejemplo anterior, el Libro define un tipo que entiende estos mensajes:

- precio() y precio(_precio): implícitos por la definición de la propiedad precio
- esCaro()
- aumentar(aumento)

Pero además de la clase, nosotros podemos definir otros tipos para el mismo objeto. Por ejemplo, podemos definir que un libro, un teléfono celular o un auto comparten en común un tipo: todos nos saben decir si son caros, independientemente de cómo lo implementen (podríamos no tener referencias al precio).

Repasamos entonces: ¿de qué tipo es un objeto? Puede ser de muchos, “El Principito” puede ser

- un libro
- un objeto que sabe decirnos si es caro
- o un objeto de una clase Wollok genérica, que sabe mostrarse por consola

Cada una de estas características define un **tipo**.

Los tipos permiten encontrar similitudes entre clases que aparentemente no tienen conexión, como veremos en el siguiente ejemplo.

6 Polimorfismo con clases

Además de libros, en el local agregamos la venta de DVD's y láminas. De todos los productos se conoce su precio.

- En el caso de los DVD el precio de venta es el precio por minuto * cantidad de minutos del dvd. Además, se sabe que es caro si la cantidad de minutos es mayor a 100.
- En cambio en las láminas el precio es el ancho * alto * precio base del material con el cual está pintada (ej.: acuarela, óleo, crayón, etc). Y se sabe que es cara si el precio base del material es mayor a 50 pesos.

Si bien todos los productos definen un método precio(), el cálculo del precio es diferente para cada producto, ya sea libro, DVD o lámina. Como el comportamiento es diferente, necesitamos clases diferentes. Por lo tanto vamos a crear la clase DVD y la clase Lámina en Wollok así:

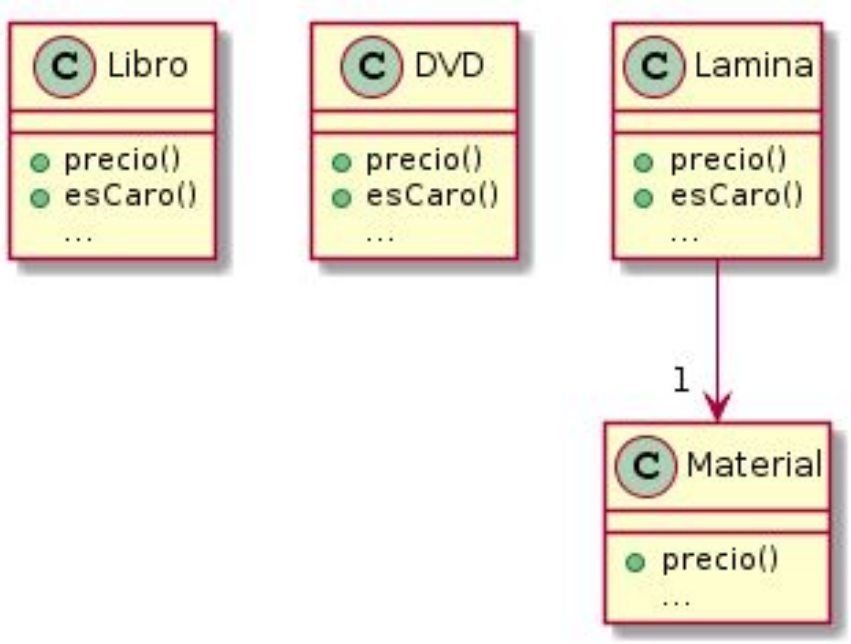
```
class DVD {
	var property precioMinuto = 10
	var property cantidadMinutos = 90

	method precio() = precioMinuto * cantidadMinutos
	method esCaro() = cantidadMinutos > 100
}

class Lamina {
	var property ancho = 29.7
	var property alto = 42
	var property material = new Material()

	method precio() = alto * ancho * self.precioBase()
	method esCaro() = self.precioBase() > 50
	method precioBase() = material.precio()
}
```

`self.precioBase()` se refiere al objeto receptor del mensaje: una instancia de la clase Lamina.

Además necesitamos en nuestro modelo una clase Material de la cual conocemos su precio.

```
class Material {
	var property precio = 5
}
```

El diagrama de clases nos quedaría[2]:



Vamos a crear las instancias de la clase DVD y de la clase Lamina. Ambos objetos entienden el mensaje precio().

```
const jurassicPark = new DVD()
const laGioconda = new Lamina()
```

De esta manera, ya están los objetos instanciados correctamente y se les puede enviar a cualquiera de ellos el mensaje precio().

```
jurassicPark.precio()
laGioconda.precio()
```

aunque el comportamiento de cada método precio() sea diferente.

[2] El lector podrá preguntarse: ¿no falta documentar las variables? ¿y algunos métodos?
La recomendación que hacemos es: no todo tiene que escribirse en el diagrama, solo lo que consideremos importante. Al fin y al cabo, la implementación puede verse en el código mismo. El diagrama está mostrando relaciones y comportamiento desde otro punto de vista.

7 Tipos y polimorfismo

Hemos visto de esta manera que una lámina, un DVD y un libro son polimórficos, porque aunque cada uno conserva su propia implementación, quien le envía mensajes no nota la diferencia y es el que aprovecha ese beneficio:

```
class Comprador {
	method comprar(algo) {
		if (algo.esCaro()) {
			...
```

Para hacer la pregunta esCaro(), la referencia a *algo* puede apuntar a

- un libro
- un DVD
- una lámina
- o a cualquier otro objeto conocido (wko) o de una clase que implemente el método esCaro()

El comprador, entonces, ni siquiera necesita conocer al objeto exacto al que le está hablando. Sabe solamente que le puede enviar el mensaje esCaro(), y eso

- abre el juego a que aparezcan nuevos objetos sin necesidad de modificar al Comprador (por eso decimos que hay un bajo acoplamiento entre ambos componentes: comprador por un lado y producto por el otro)
- permite dividir tareas: una persona puede codificar al Comprador y otra/s a los productos
- el polimorfismo entre estos productos se da en el contexto de la compra: esto no implica que se comportan por igual en todos los demás contextos. Ciertamente la forma de inicializar una lámina, un DVD y un libro es totalmente diferente. Además, en este modelo, los libros pueden aumentar su precio pero los demás elementos no.

8 Repaso: objetos anónimos, wko y clases

Ahora que incorporamos a las clases como una forma más de representar conceptos, podemos preguntarnos ¿cuándo conviene modelar a través de objetos anónimos, cuándo usar wko y cuándo clase?

- los **objetos anónimos** tienen un alcance acotado, por ejemplo un test, o la resolución de un requerimiento (un método y todo el encadenamiento de mensajes que sale de ese método). Lo mismo pasa cuando necesitamos construir un bloque para pasárselo a filter, o al map de una colección: el

objeto que representa al bloque no tiene nombre y no nos importa que lo tenga, existe para resolver el requerimiento puntual.

```
method taxistasBuenos() =
      taxistas.filter { taxista => taxista.esBueno() }
      // las llaves encierran un objeto anónimo que representa el
      // criterio de selección de un taxista
```

- **wko**: son objetos conocidos, porque están representando un concepto de negocio dentro de la aplicación. Esto ocurre cuando un objeto tiene un comportamiento específico, y nos interesa modelarlo en forma separada de otros objetos.

- por último, las **clases** son importantes cuando se que existen múltiples objetos que comparten comportamiento y no tiene sentido que los nombre por separado: el viaje que hice ayer en colectivo, se parece mucho al viaje de la semana pasada. Si solo difieren en la información que guardan las referencias, el comportamiento se debe ubicar en un solo lugar para no repetir la misma idea una y otra vez.

9 Resumen

Hemos presentado a la clase como un concepto que permite agrupar los objetos con el mismo comportamiento y la misma estructura, más allá de que cada instancia guarde en cada referencia su propio juego de valores. Todo objeto debe pertenecer a una clase durante todo su ciclo de vida, y a su vez una clase tiene diferentes tipos, en base a los mensajes que define. Cuando dos objetos de diferente clase comparten el mismo tipo, son polimórficos en ese contexto: un observador le envía mensajes a los dos y no percibe cambios, aunque cada clase define un método propio.