Paradigma
Orientado a Objetos

Módulo 14:
Mutabilidad.
Igualdad e identidad.

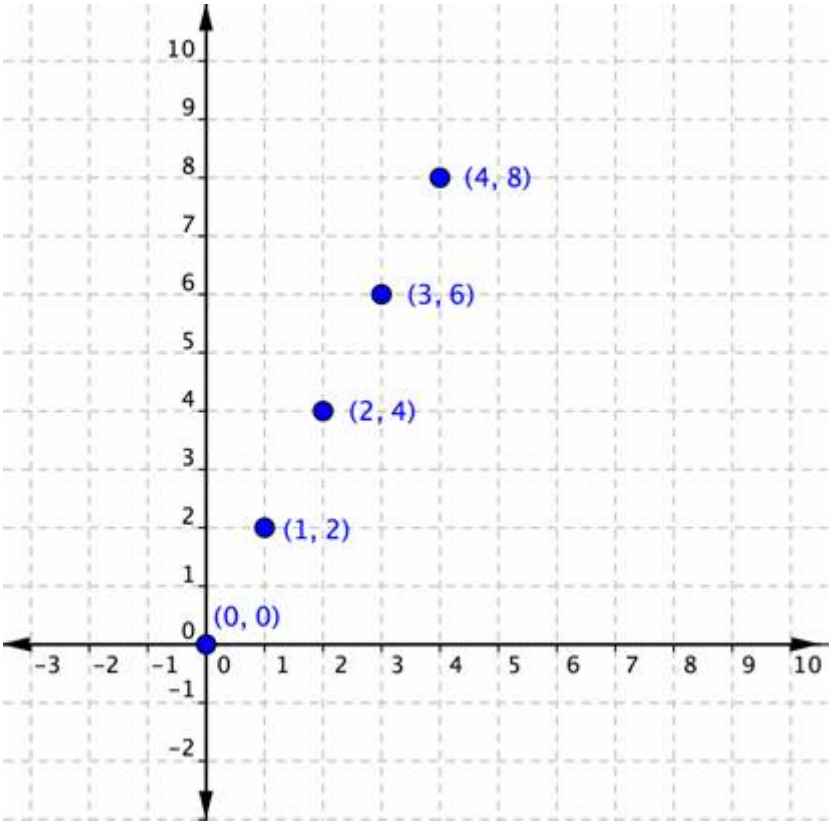
por Fernando Dodino
Alfredo Sanzo
Versión 3.0
Octubre 2024

Distribuido bajo licencia Creative Commons Share-a-like

Indice

1 Mutabilidad / inmutabilidad

Si queremos modelar un punto en un eje de coordenadas:



Tenemos dos decisiones de diseño posible:

- hacer que el objeto sea **mutable**, definiendo setters para las propiedades *x* e *y*
- construir un objeto **inmutable**. El punto, una vez construido, no puede variar: representa una ubicación en el plano y no puede representar otro punto más que ése.

¿Qué pasa si queremos sumar dos puntos? Se termina construyendo un punto nuevo...

```
class Point {
  const property x
  const property y

  method +(otroPoint) =
    new Point(x = x + otroPoint.x(), y = y + otroPoint.y())
}
```

Esta misma estrategia podemos adoptar para

- los números: tenemos un objeto 2 y otro que representa al 3, si los sumo el 2 no “cambia” a 5, el 5 es un nuevo objeto
- los strings, que son inmutables: cuando quiero concatenar dos strings, genero uno nuevo
- los booleanos, ya que en realidad existe un solo true y un solo false

1.1 Value objects

En general, todos los objetos que describimos recién entran en la categoría de *Value Objects*: son objetos que representan un valor de nuestro dominio. Otros ejemplos posibles podrían ser: objetos que representan un color, como el rojo, un objeto que modela un mail, un objeto que representa una figura bidimensional (sería un value object construido como una lista de puntos), una fecha, etc.

1.2 Motivación

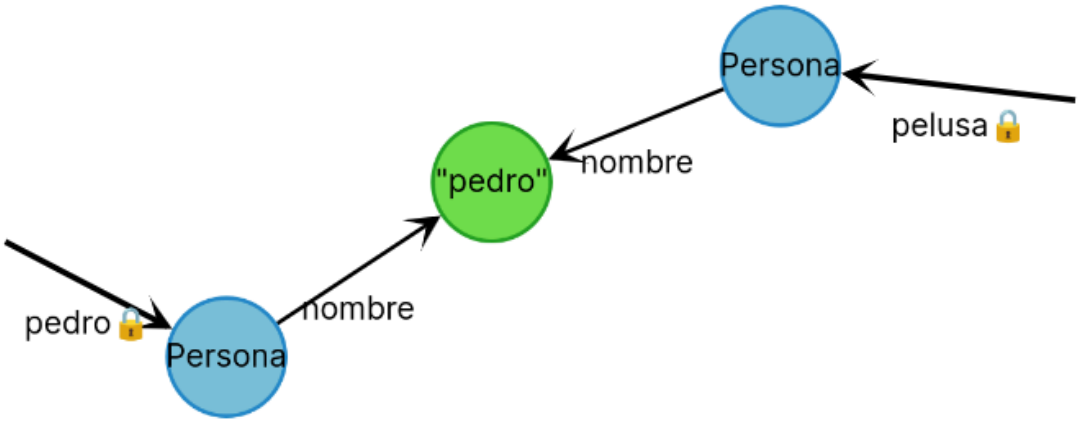
¿Por qué queremos tener objetos inmutables? Porque son objetos que no tienen efecto colateral: más allá de que el paradigma lo soporte, yo elijo no trabajar con este concepto, reforzando la idea de que el paradigma está en quienes desarrollan.

Al no tener efecto colateral

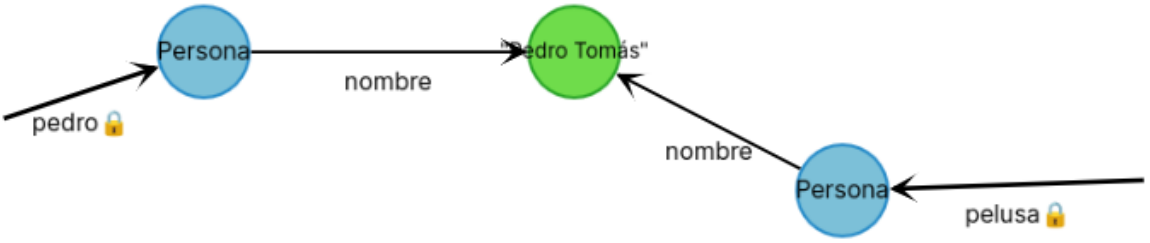
- el testing se simplifica, porque entran en juego una menor cantidad de situaciones y contextos
- es más fácil compartir los objetos en forma concurrente, porque sabemos que nadie puede hacer modificaciones a ese objeto

Consideremos por ejemplo dos personas que tienen el mismo nombre: Pedro.

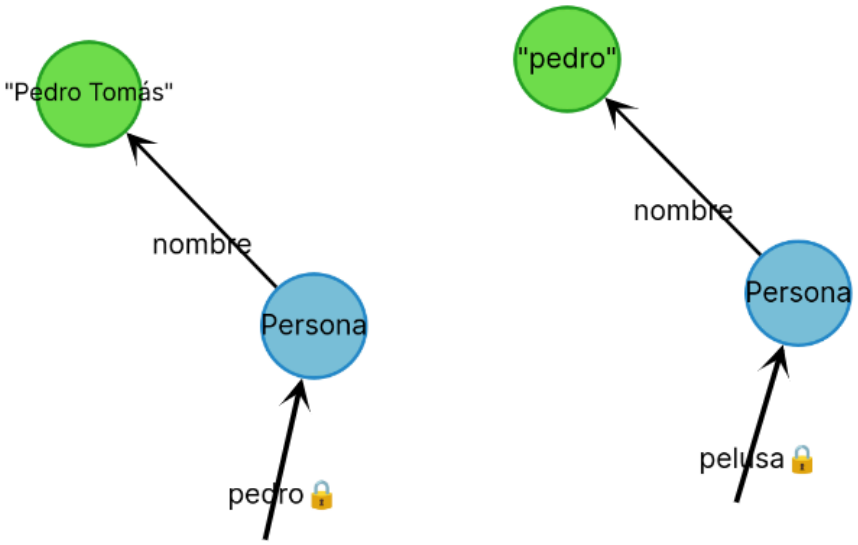
```
class Persona {
  var property nombre
}

const pedro = new Persona(nombre = "pedro")
const pelusa = new Persona(nombre = "pedro")
```



Si los Strings fueran mutables, al enviar el mensaje
`pedro.nombre("Pedro Tomás")`
¡estaríamos cambiando la referencia nombre de `pelusa`!



Pero eso no es lo que sucede:

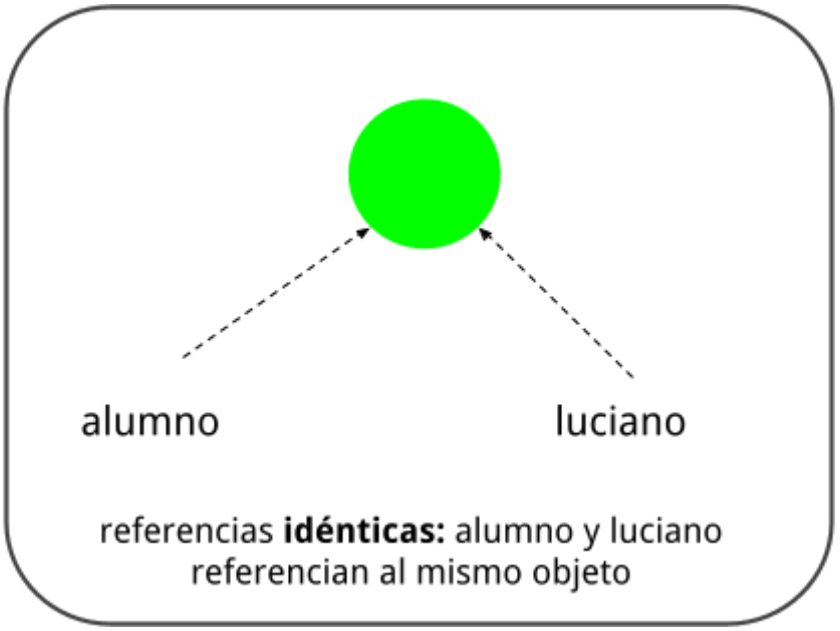


2 Igualdad e identidad

Otro concepto importante en el manejo de referencias es diferenciar la igualdad vs. la identidad.

2.1 Identidad

Si tenemos dos referencias idénticas, esto significa que están apuntando al mismo objeto.



Consideremos una clase que modela un Mail

```
class Mail {
  const property asunto
  const property emisor
  const property receptor
}
```

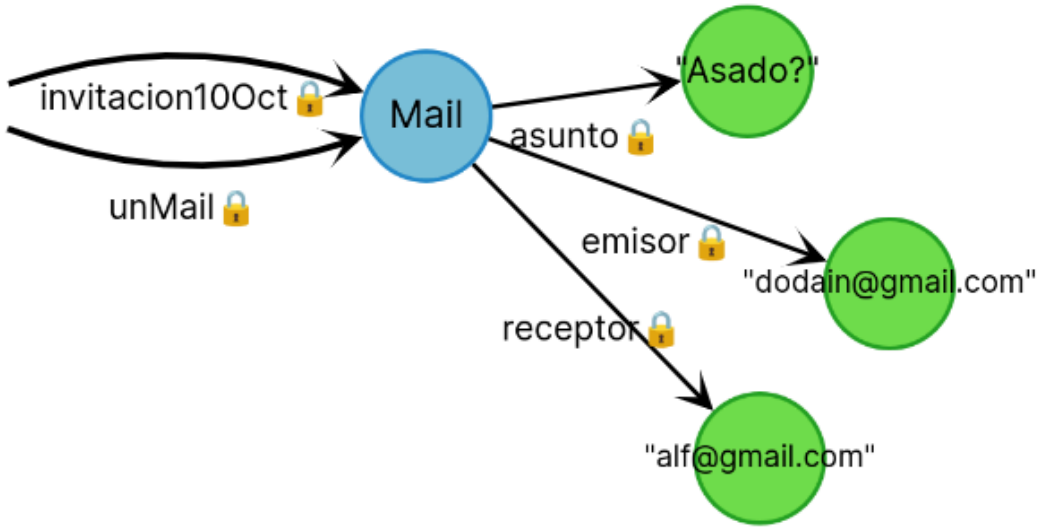
Genero una instancia del mail

```
const invitacion10Oct = new Mail(asunto = "Asado?",
      emisor = "dodain@gmail.com", receptor = "alf@gmail.com")
```

Si escribo la siguiente línea

```
const unMail = invitacion10Oct
```

en el diagrama se ve claramente que ambas referencias apuntan al mismo objeto.



Esto es lo que se conoce como **identidad** y podemos utilizar el mensaje === en Wollok para preguntar si dos referencias son idénticas:

```
> unMail === invitacion10Oct
✓ true
```

2.2 Igualdad

En la mayoría de los casos estaremos bien con esta definición. Pero a veces tendremos que cambiar la estrategia para determinar si dos referencias están *representando* al mismo objeto, aun cuando no se trate exactamente del mismo objeto. Este concepto se llama **igualdad**.

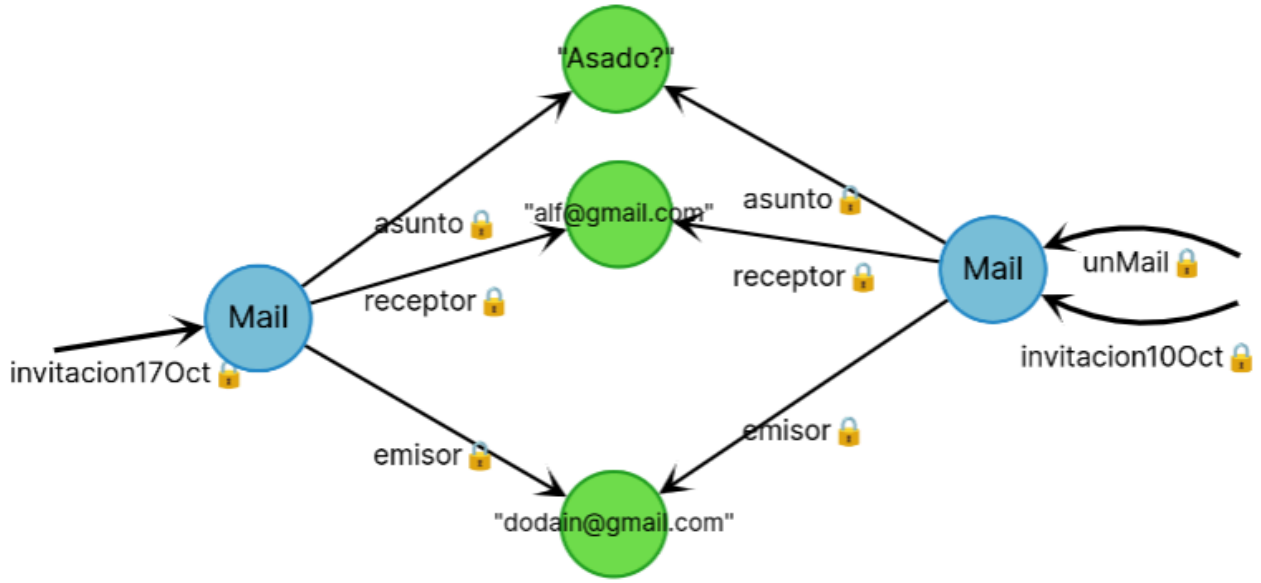
Si otro día mandamos un mail que tiene el mismo asunto, y sabemos que emisor y receptor son los mismos, ¿no es en definitiva **el mismo mail**?

```
const invitacion17Oct = new Mail(asunto = "Asado?",
          emisor = "dodain@gmail.com", receptor = "alf@gmail.com")
```

El mensaje que nosotros usamos para preguntar si son iguales en Wollok es ==:

```
> invitacion17Oct == invitacion10Oct
✓ false
```

Por defecto, dos referencias son iguales si apuntan al mismo objeto:



Hasta ahora tenemos:

Consulta	Significado
`> invitacion17Oct === invitacion10Oct` ✓ `false`	¿Son idénticas las referencias invitación del 10 y la del 17? No, porque apuntan a objetos distintos
`> unMail === invitacion10Oct` ✓ `true`	¿Son idénticas las referencias invitación del 10 y unMail? Sí, porque apuntan al mismo objeto
`> invitacion17Oct == invitacion10Oct` ✓ `false`	¿Son iguales la invitación del 10 y la del 17? No. EPA. ¿Por qué? **Yo quería que, si representan lo mismo, sean iguales, aunque no sean el mismo objeto.**

Pará... Entonces, por defecto la igualdad está funcionando igual que la identidad. ¿Entonces para qué me sirve tener igualdad e identidad por separado? Dijimos que queremos definir cuando **dos objetos representan lo mismo**. Bueno, si redefinimos el método *equals* o == logramos lo que queremos. Por ejemplo, dos mails pueden ser iguales si tienen el mismo asunto, el mismo emisor y el mismo receptor. Redefinimos entonces el mensaje == en Mail:

```
class Mail {
  const property asunto
  const property emisor
  const property receptor

  override method == (otroMail) =
    asunto == otroMail.asunto() &&
    emisor == otroMail.emisor() &&
    receptor == otroMail.receptor()
}
```

Si recargo la sesión, ya puedo preguntar si las referencias son idénticas e iguales:

```
> invitacion17Oct === invitacion10Oct
✓ false
> invitacion17Oct == invitacion10Oct
✓ true
```

Lo cual tiene sentido porque aun cuando no referencien al mismo objeto **representan al mismo objeto.**

Es por esto que queremos diferenciar igualdad (==) de identidad (===). La igualdad nos da *un punto de modificación, un punto de acuerdo*, para **controlar nosotros** qué significa la igualdad de objetos.

2.3 ¿Igualdad o identidad?

Por lo general, la recomendación es preguntar siempre por igualdad. Es raro tener que verificar que las referencias sean idénticas, porque es un problema que maneja la Virtual Machine.

Cuáles son los casos borde de esta afirmación:

- a veces, quienes están en proceso de optimizar la VM o un programa muy específico, se encuentran con que comparar por == es costoso: tengo que comparar todos los valores. Entonces, para aquellos objetos para los que me pueda asegurar que exista solo una instancia, podría usar la identidad (el mensaje ===) para comparar, que es mucho más rápida. Un ejemplo concreto es el motor de Wollok Game, que necesita tener optimizaciones para poder procesar los cambios de los elementos visuales varias veces por segundo.

- también es común que los objetos puedan vivir en dos ambientes. Esto se da en aplicaciones distribuidas, donde muchas veces hay un navegador y un servidor web. Ahí, como directamente estamos en dos máquinas virtuales distintas (incluso pueden ser de distintas tecnologías), cuando se pase un objeto de un ambiente al otro, voy a tener un cliente, y a la "copia" del cliente que me vino por red. Si quiero compararlos, la identidad no me sirve, porque de prepo existen dos objetos. Necesito la igualdad para encontrarla, y tendré que diseñar de qué manera me doy cuenta de que dos referencias apuntan al mismo cliente.

3 Resumen

Hemos visto que los objetos pueden ser mutables, en cuyo caso podemos cambiar las referencias, o ser inmutables (una vez construidos no pueden cambiar sus referencias, carecen así de efecto colateral). Esto nos permite compartirlos sin necesidad de estar cuidando las modificaciones en forma concurrente, y también facilita su testeo. Los objetos que modelan valores (como los strings, números, fechas, etc.) son buenos ejemplos de objetos que nos conviene que sean inmutables.

Cada objeto tiene una identidad que lo diferencia de los demás objetos del ambiente. En general preguntamos por referencias idénticas cuando estamos hablando de objetos de nuestro dominio, pero cuando tenemos strings, números, domicilios, puntos, booleanos no es conveniente preguntar si dos referencias apuntan exactamente al mismo *value object*, sino que hacemos la pregunta por igualdad (*==*, *equals*).