****

**Paradigma**

**Orientado a Objetos**

**Módulo 15:**

**Mutabilidad.**

**Igualdad e identidad.**

**por Fernando Dodino**

**Versión 2.1**

**Octubre 2017**

Distribuido bajo licencia [Creative Commons Share-a-like](https://creativecommons.org/licenses/by-sa/4.0/legalcode)

**Indice**

[1 Mutabilidad / inmutabilidad](#_iryaqk9d08i2)

[1.1 Value objects](#_4ynojb5d0flz)

[1.2 Motivación](#_4y8ifvxeqet0)

[2 Igualdad e identidad](#_hryrt6t60c2h)

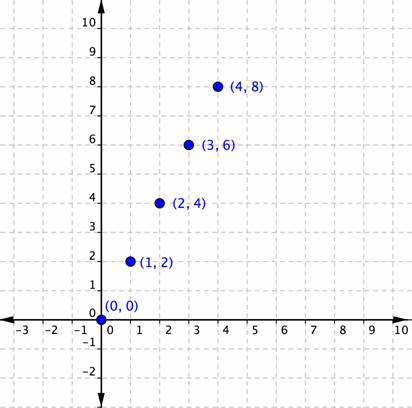
[2.1 Identidad](#_i7yfqemx2py5)

[2.2 Igualdad](#_7ht5alpzkn71)

[3 Resumen](#_52oerlg2pamq)

# 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

**constructor**(\_x, \_y) {

x = \_x

y = \_y

}

**method** +(otroPoint) =

**new** Point(x + otroPoint.x(), 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 existen dos well known objects: true y 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 a partir de otros value objects, como el punto definido anteriormente), una fecha, etc.

## 1.2 Motivación

¿Por qué querríamos 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.



Si los Strings fueran mutables, al enviar el mensaje

ameo.nombre("Pedro Tomás")

¡estaríamos cambiando la referencia de wachin!



Pero eso no es lo que sucede... afortunadamente.



# 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.



Esto en Wollok se representa mediante el mensaje ===

Consideremos la clase Persona...[[1]](#footnote-0)

**class** Persona {

**var** nombre

**var** domicilio

**constructor**(\_nombre, \_domicilio) {

nombre = \_nombre

domicilio = \_domicilio

}

**method** nombre() = nombre

**method** domicilio() = domicilio

}

Nuestro primer test es sencillo: utilizando la asignación, ambas referencias apuntan al mismo objeto.

**import** personas.\*

**test** "si asigno una referencia a otra ambas apuntan al mismo objeto" {

**const** alumno = **new** Persona("luciano", "Centenera 83")

**const** luciano = alumno

assert.that(alumno === luciano)

}

## 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**.

Por defecto, dos objetos son iguales si son el mismo objeto. Pero esa definición está sujeta a cambios, si redefinimos el método equals / ==. Por ejemplo, dos personas podrían ser iguales si tienen el mismo nombre.

Ahora bien

* ¿por qué tendría en el ambiente dos alumnos con el mismo legajo?
* ¿o dos personas con el mismo DNI?
* ¿o dos materias que se llaman igual?

En una solución donde yo tengo una única VM, no tiene sentido. Ahora bien, cuando tenemos una aplicación comercial tenemos que relajar un poco esta cuestión

* por ejemplo, cuando la información sobre una persona no está en una VM específica, sino que se almacena en un medio persistente. Cada sesión va haciendo cambios sobre los objetos de dominio y los persiste en algo externo a la VM. Entonces, cuando una sesión de usuario necesita recuperar la información de un cliente, va al medio persistente y reconstruye el objeto Cliente en base a la información almacenada. El objeto cliente tiene un ciclo de vida corto dentro del ambiente, termina actuando como un *value object*: es el objeto que representa a un cliente “en un momento dado” (por ejemplo cuando paga una factura, o cuando quiero aumentarle la deuda).
* cuando tengo una aplicación distribuida en nodos cliente y servidor. Es más, al tener varias sesiones de usuario, tengo n clientes y al menos 1 servidor. En cada uno de los ambientes (cliente o servidor) puedo tener objetos cliente, entonces la idea de identidad como la conocimos pierde sentido, tengo que buscar otro criterio para identificar tal o cual cliente mientras lo voy pasando de un ambiente a otro.

Pero volvamos a nuestro ejemplo didáctico, donde existe una sola VM y no tenemos el concepto de sesiones de usuario todavía. Codificamos entonces la clase Persona **sin redefinir el equals**, después de todo es probable que encontremos varias personas que se llamen igual pero no por eso representarán la misma persona.

Ahora bien, una persona vive en un domicilio. ¿Cómo representamos esa abstracción? Mediante un String. Entonces tenemos a Chiara y Melina que viven en ‘Nazca 3143’. Pero como los Strings los genera la VM, no puedo asegurar que los domicilios de ambas personas referencien exactamente a la **misma** instancia. Entonces cuando pregunto si Chiara y Melina viven juntas, no está bueno confiar en el ===, es preferible preguntar por equals.

Si definimos el siguiente comportamiento en Persona:

**method** vivenJuntosCon(\_otraPersona) =

domicilio === \_otraPersona.domicilio()

El siguiente test falla:

**test** "dos objetos domicilio distintos pero con la misma dirección son iguales" {

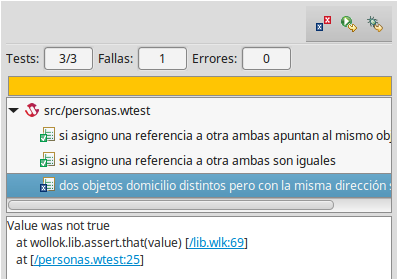
**const** chiara = **new** Persona("Chiara", "Nazca 3143")

**const** melina = **new** Persona("Melina", "Nazca 3143")

assert.that(chiara.vivenJuntosCon(melina))

}

Gráficamente





Eso puede verse enviando este mensaje en el test anterior:

console.println(chiara.domicilio().identity()) // 2095573052

console.println(melina.domicilio().identity()) // 847018986

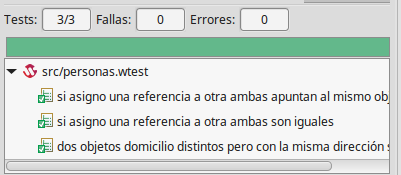
El mensaje identity() devuelve un número que podemos usar para diferenciar un objeto de otro en el ambiente.

Ahora bien, si reemplazamos el uso de === en la definición por ==, ya no dependemos de que exista un único String que represente al domicilio:

**method** vivenJuntosCon(\_otraPersona) =

domicilio == \_otraPersona.domicilio()

Y el mismo test que antes fallaba, ahora pasa perfectamente...



En general, cuando estamos hablando de objetos de nuestro dominio (los que nosotros diseñamos) es lógico pensar que cada objeto va a representar un concepto dentro de mi ambiente: no habrá dos objetos para el mismo alumno, ni dos objetos para el mismo cliente, siempre y cuando yo tenga una aplicación con un único ambiente. En cambio, cuando tengamos números, strings, fechas, los *value objects*, tenemos que estar atentos a no esperar que se cumpla la identidad sino la igualdad. Entonces dos personas habrán nacido el mismo día si coinciden sus fechas de nacimiento, y dos empleados ganarán lo mismo si sus sueldos son iguales.

# 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, 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).

1. El ejemplo puede descargarse en <https://github.com/wollok/igualdad-identidad-domicilios> [↑](#footnote-ref-0)