****

**Paradigma**

**Orientado a Objetos**

**Módulo 09:**

**Herramientas de instanciación**

**por Fernando Dodino**

**Versión 1.0**

**Enero 2018**

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

**Indice**

[1 Formas de instanciar un objeto](#_h79gy0rw69hc)

[2 Introducción a constructores](#_ceqetiyv274e)

[3 Inicialización de un objeto](#_ixn8muisbtx)

[3.1 Valores y constructor por defecto](#_axmlczg29dq7)

[3.2 Constructor por defecto + asignación mediante setters](#_ngodxsmdk2wr)

[3.3 Trabajar con referencias constantes y definir un constructor específico](#_yq72du3xtriu)

[3.4 Parámetros nombrados](#_vj2ax6a4f7xh)

[4 Resumen](#_v6qm3b2pe77n)

## 

# 1 Formas de instanciar un objeto

Los objetos anónimos se creaban en forma programática:

**const** alumno = **object** {

**method** estudia() = **true**

}

Mientras que los wko se instanciaban solos en el momento en que uno ejecuta la consola REPL, o bien un test, o un programa.

También están los literales de Wollok: objetos que tienen una sintaxis propia para construirse:

2 // permite referenciar al número 2

"hola" // construye un String cuyo valor es "hola"

[7] // construye una lista con un único elemento: el número 7

¿Cómo instanciamos una fecha?

>>> new Date()

Pero también podemos instanciarlo con valores:

>>> new Date(17, 8, 2017)

# 2 Introducción a constructores

Para poder crear una instancia de una clase, necesitamos que esa clase defina al menos un constructor, un mecanismo que nos dice qué debe ocurrir cuando instanciamos un objeto.

Por defecto, cuando creamos una clase no debemos escribir ningún constructor, porque viene "de fábrica" con un constructor sin parámetros.

Entonces, si tenemos esta definición

**class** Ave {

**var** energia = 0

**method** volar(kms) { energia = energia - (kms \* 5) }

}

sabemos que para crear un ave en la consola podemos hacerlo de esta manera:

**>>> const cormoran = new Ave()**

¿Cómo queda inicializado el cormorán? En este caso hay una sola referencia, energía, que tiene como valor 0. Podemos enviar el mensaje volar sin problemas, el objeto está correctamente construido.

A continuación vamos a analizar diferentes casos de cómo podemos construir un objeto y qué sucede con sus respectivas referencias.

# 3 Inicialización de un objeto

## 3.1 Valores y constructor por defecto

Como vimos en el ejemplo del Ave, cuando el objeto es lo suficientemente simple, nos alcanza con tener variables inicializadas en su definición y dejar el constructor por defecto que Wollok provee.

## 3.2 Constructor por defecto + asignación mediante setters

Cuando las referencias de un objeto son específicas, una opción consiste en publicarlas como propiedades (o escribir setters manualmente) y dejar el constructor por defecto sin parámetros:

**class** Persona {

**var** **property** nombre

**var** **property** apellido

**method** presentarse() =

"Hola, me llamo " + nombre + " " + apellido

}

Entonces en un test construimos una persona de esta manera:

**test** "saludar a una persona" {

**const** lito = **new** Persona()

lito.nombre("Jose")

lito.apellido("Guerrero")

assert.equals("Hola, me llamo Jose Guerrero",

lito.presentarse())

}

La ventaja de esta solución es que requiere muy poco código en la definición de la persona. También es simple construir una persona en el test.

Las desventajas

* para inicializar 10 personas, hay 30 líneas de código repetitivas
* en algún momento el objeto persona que estamos construyendo queda inconsistente (tiene referencias sin inicializar), ¿y cuál es el problema?

**test** "una persona se presenta" {

**const** lito = **new** Persona()

lito.nombre("Jose")

lito.apellido().length() // Crash boom bang!

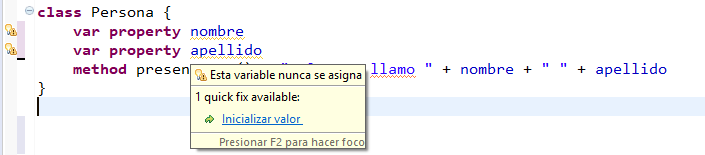
lito.apellido("Guerrero")

assert.equals("Hola, me llamo Jose Guerrero", lito.presentarse())

}

Ah, ah, ah.

La buena noticia es que el IDE de Wollok nos alerta cuando hay variables que no se inicializan con un valor por defecto ni en los constructores.



Por eso una alternativa posible es definir un valor por defecto. En el caso de las referencias var, esto es opcional mientras que en el caso de las referencias const esto es algo obligatorio.

## 3.3 Trabajar con referencias constantes y definir un constructor específico

Podemos definir también a la persona con referencias constantes, lo que requiere que escribamos un constructor donde pasemos dichas referencias[[1]](#footnote-0).

**class** Persona {

**const** **property** nombre

**const** **property** apellido

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

nombre = \_nombre

apellido = \_apellido

}

**method** presentarse() = "Hola, me llamo " + nombre + " " + apellido

}

En el test la instanciación de una persona requiere una sola línea y nunca queda en estado inconsistente:

**test** "una persona se presenta" {

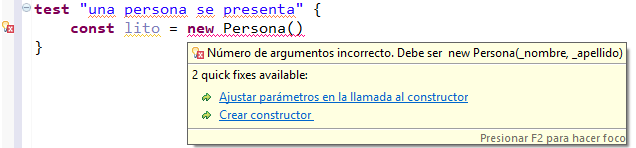
**const** lito = **new** Persona("Jose", "Guerrero")

assert.equals("Hola, me llamo Jose Guerrero",

lito.presentarse())

}

Es importante recalcar que al definir un constructor ya no contamos con el constructor por defecto sin parámetros:

****

a menos, claro está, que lo definamos explícitamente.

Al definir constructores específicos, ya no es obligatorio definir valores por defecto en las referencias var o const, mientras las inicialicemos en todos los constructores que definamos.

## 3.4 Parámetros nombrados

Wollok permite nombrar las referencias al construir el objeto, lo que ayuda a darle más claridad a la instanciación. Por ejemplo, queremos modelar la presentación de un recital, que define

* en qué lugar se realizará (puede cambiar)
* qué músicos tocarán
* cuál será la fecha (puede reprogramarse)
* el valor de la entrada general
* y cuántas entradas se vendieron hasta el momento

Si pensamos en tener un constructor con varios parámetros:

**class** Presentacion {

**var** **property** lugar

**const** **property** musicos

**var** **property** valorEntrada

**var** **property** fecha

**var** **property** entradasVendidas

**constructor**(\_lugar, \_musicos, \_fecha, \_valorEntrada, \_entradasVendidas) {

lugar = \_lugar

musicos = \_musicos

fecha = \_fecha

valorEntrada = \_valorEntrada

entradasVendidas = \_entradasVendidas

}

}

La instanciación de una presentación a un show puede ser poco reveladora en el test:

**const** seruEnLuna = **new** Presentacion(lunaPark, [seruGiran],

**new** Date(15, 10, 2018), 2000, 180)[[2]](#footnote-1)

Utilizamos entonces la técnica de parámetros nombrados, para lo cual primero debemos configurar valores default para las referencias const:

**class** Presentacion {

**var** **property** lugar

**const** **property** musicos = []

**var** **property** valorEntrada

**var** **property** fecha

**var** **property** entradasVendidas

}

Y ahora sí instanciamos la presentación de un show:

**const** seruEnLuna = **new** Presentacion(

lugar = lunaPark,

musicos = [seruGiran],

fecha = **new** Date(15, 10, 2018),

valorEntrada = 2000,

entradasVendidas = 180)

Estos parámetros revelan más claramente cuáles son las referencias a inicializar. Además, mientras que al generar el constructor estamos obligados a respetar el orden:

**const** seruEnLuna = **new** Presentacion(180, lunaPark, [seruGiran],

**new** Date(15, 10, 2018), 2000)

 **Se está asignando el valor 180 al lugar, lunaPark al conjunto de músicos, etc.**

con los parámetros nombrados podemos ingresar los valores en cualquier orden:

**const** seruEnLuna = **new** Presentacion(

entradasVendidas = 180,

lugar = lunaPark,

musicos = [seruGiran],

fecha = **new** Date(15, 10, 2018),

valorEntrada = 2000)

Por otra parte

* algunos encuentran demasiado verbosa esta variante (hay que escribir más aunque el IDE ayuda a que eso no ocurra)
* la responsabilidad de inicializar bien depende de quien instancia a la presentación (mientras que si definimos nuestro constructor eso queda asegurado por los parámetros que exige). Por lo tanto, definir valores por defecto en la declaración de las referencias vuelve a ser importante para evitar errores en el envío de mensajes. Por ejemplo, en esta línea

**const** seruEnLuna = **new** Presentacion(

lugar = lunaPark,

musicos = [seruGiran],

fecha = **new** Date(15, 10, 2018),

valorEntrada = 2000)

Está faltando inicializar las entradas vendidas. Cualquier mensaje que enviemos al objeto representado por seruEnLuna que involucre a las entradas vendidas producirá un error. En cambio si proveemos un valor por defecto en la definición

**class** Presentacion {

...

**var** **property** entradasVendidas = 0

Ya no tendremos problemas en el envío de mensajes. Esto mismo en las referencias constantes es obligatorio.

En general, el formato para utilizar los parámetros nombrados es:

**new** ClaseAInstanciar(

referencia1 = valor1,

referencia2 = valor2,

...,

referenciaN = valorN)

Las referencias no necesariamente deben ser propiedades (ni es necesario generarles *setters*), sí deben estar definidas como atributos válidos y pueden tener o no valores por defecto.

# 4 Resumen

A lo largo de este capítulo hemos visto diferentes formas de instanciar un objeto y de inicializar las referencias de manera de poder garantizar su uso. Partiendo desde la opción más simple a la más compleja, podemos enumerar las siguientes técnicas:

1. definir valores en la declaración de las variables y aprovechar el constructor por defecto que ofrece Wollok
2. para las referencias que requieren valores específicos podemos asignarlas mediante setters (y continuar con el constructor sin parámetros)...
3. ...o bien definir constructores con parámetros para forzar a que el que construya el objeto pase toda la información necesaria para inicializarlo
4. por último, los parámetros nombrados le dan flexibilidad y claridad en la llamada al constructor, a cambio de escribir los nombres de las referencias que queremos inicializar.

1. Como el nombre y el apellido son referencias a String, que son objetos inmutables, estamos modelando a la persona como un objeto inmutable también. Más adelante profundizaremos sobre estas cuestiones. [↑](#footnote-ref-0)
2. Tanto lunaPark como seruGiran son wko creados en el archivo .wlk [↑](#footnote-ref-1)