# Paradigmas de Objetos

La principal abstracción de nuestro nuevo paradigma son los objetos, que representan distintos tipos de entidades.

Un objeto es algo que puedo representar a través de una idea, un concepto. Tiene entidad.

Un sistema en el paradigma orientado a objetos es: “Un sistema es un conjunto de objetos que se envían mensajes para alcanzar un determinado objetivo”

¿Cuál era la definición de sistema según la Teoría General de Sistemas? “Conjunto de partes que se relacionan para un objetivo común” Bueno, al menos sabemos que vamos con una definición congruente...

## Mensaje

¿Cómo interactúan los objetos entre sí? A través de los mensajes. ¿Cómo envío un mensaje a otro objeto?

* primero lo tengo que conocer (más adelante veremos cómo)
* cuando lo conozco, sé qué mensajes le puedo mandar
* ¿se cómo lo hace? no, no quiero saberlo, porque saberlo me ata a él (en términos de diseño decimos “me acopla”) y si yo asumo ciertas cosas en base a cómo lo hace, eso aumenta la posibilidad de tener que modificar cosas si el objeto al que le mando el mensaje cambia.

## Ambiente

La palabra ambiente es conocida en otras tecnologías como

* imagen, en Smalltalk
* virtual machine, para Java y otras tecnologías similares

## Mensaje y método

Mensaje es lo que el objeto emisor le envía como orden al receptor. El emisor no se entera de cómo se resuelve el mensaje, sólo lo pide (el qué y no el cómo, ¿a qué suena eso?). El receptor recibe el mensaje y se ejecuta un método (porción de código).

### Method lookup

La estrategia que utilizan los lenguajes para resolver dónde está el código de un método al enviar un mensaje recibe el nombre de method lookup. En nuestro primer caso (el más simple) el código a ejecutar de un mensaje se busca en un método del objeto receptor de dicho mensaje.

# Breve introducción a Wollok

Wollok es un lenguaje de programación con fines didácticos

* no obliga al desarrollador a definir tipos para los objetos
* es interpretado
* no necesitas tener un programa editor de texto y una línea de comando para correr los programas/tests sino que provee un entorno integrado de desarrollo o IDE donde trabajar.

## Accessors

Como vimos en el párrafo Estado de un objeto, no es aconsejable que quien use a pepita manipule directamente sus variables, sino que debe enviarle mensajes que accedan o modifiquen las referencias a los objetos que conoce. Para ello existe cierto tipo de mensajes llamados accessors, cuyo fin es publicar la referencia (getter) o modificar dicha referencia (setter).

El getter se define como un método de una sola línea, que devuelve algo.

El setter no, es un método que modifica el estado interno del objeto, no tiene necesidad de devolver nada.

## Referencias en Wollok

Wollok tiene dos maneras de definir referencias

* variables (var): es una referencia que puede cambiar el objeto al que apunta
* valores (const): es una referencia fija a un objeto, que no puede cambiar (una vez inicializada no es válida la operación de asignación)

## Relaciones bidireccionales

Mantener una relación bidireccional

* es cómodo, porque cualquiera de los dos objetos puede enviar un mensaje a otro
* pero requiere mantener ambas referencias sincronizadas, para evitar inconsistencias

Una alternativa puede ser que uno de los objetos sea el responsable de actualizar la otra referencia:

## Identidad

En un sistema con objetos, cada objeto sabe que es él y ningún otro objeto más, por eso es el responsable de responder ante un mensaje. En el ambiente cada objeto tiene su propia identidad. En un sistema orientado a objetos es frecuente tener diferentes referencias y querer determinar si estamos hablando de un objeto u otro, entonces la definición de identidad nos dice que: *Dos referencias son idénticas si apuntan al mismo objeto.*

# Introducción al Polimorfismo

El polimorfismo es una herramienta para agrupar objetos que tienen diferente comportamiento (distinto código) pero la misma interfaz. Entonces puedo enviarles mensajes sin saber cómo lo implementan, e incluso sin saber a qué objeto particular le estoy hablando: puedo cambiar la referencia sin tener que hacer cambios.

# Objetos Básicos

## Números

Los números en Wollok se representan como objetos inmutables, esto quiere decir que

* una vez inicializado un número, su estado interno no cambia
* la suma de 1 + 2 resulta en un nuevo número que representa al 3.

## Números con decimales

Es posible configurar la forma de trabajar con números que tengan decimales. En particular, se puede cambiar

* la cantidad de decimales máxima que puede admitir, por defecto son cinco
* y qué hacer en caso de recibir un número que excede la máxima cantidad de decimales permitido, por defecto lo redondea hacia arriba
* la forma de imprimir un número, por defecto eliminando los ceros no representativos

## Booleanos

Hay dos objetos booleanos representados con los literales “true” y “false”. Al igual que los números también son objetos inmutables, la expresión (true || false) devuelve un nuevo objeto true.

## Strings

Las cadenas de caracteres se delimitan con una o dos comillas.

const unString = "hola"

const otroString = 'mundo'

También son objetos inmutables (al concatenar “hola” y “mundo” tenemos un nuevo String “holamundo”).

## Fechas

Una fecha es un objeto inmutable que representa un día, mes y año (sin horas ni minutos). Se crean de dos maneras posibles:

const hoy = new Date()

// toma la fecha del día

const unDiaCualquiera = new Date(30, 6, 1973)

// se ingresa en formato día, mes y año

## Lambdas: objetos bloque

Al igual que en otras tecnologías, Wollok nos permite instanciar un objeto que representa un bloque de código, de manera de:

* poder generar referencias a dichos bloques
* pasarlos como parámetro
* elegir en qué momento ejecutar una porción de código: en un momento instanciamos un bloque de código y tiempo después
  + lo ejecutamos si se cumple una condición x,
  + ante un evento e
  + bien lo ejecutamos n veces hasta que se cumpla una condición: por ejemplo, para reprocesar una instrucción hasta que no haya errores.

### Comparación con el paradigma funcional

Más allá de la diferencia en la sintaxis, podemos hacer una comparación entre los bloques de código en Objetos y las expresiones lambdas de Haskell:

* ambas sirven para abstraer comportamiento que no interesa reutilizar en otro contexto, por eso no tienen nombre
* las expresiones lambda de Haskell solo devuelven un valor, mientras que en objetos podemos retornar un valor, o tener efecto colateral (en ese caso puede no importar el valor resultante)
* ambas son particularmente útiles para evitar hacer tareas repetitivas y subir el grado de declaratividad. Tomando los ejemplos de soluciones con map, filter y fold, tienen un menor grado de conocimiento del algoritmo y del orden que la misma solución con recursividad (en el caso de funcional) o un forEach (en el caso de objetos).

# Colecciones

La colección nos permite representar un conjunto de objetos relacionados. A primera vista una colección es un conjunto de objetos. Si la vemos con más precisión nos damos cuenta que es más preciso pensarla como un conjunto de referencias: los elementos no están dentro de la colección, sino que la colección los conoce.

## Tipos de colecciones en Wollok

A primera vista, podemos diferenciar dos diferentes tipos de colecciones en Wollok:

* los conjuntos, que modelan al conjunto matemático: no hay orden en los elementos y no puede haber elementos repetidos. Se definen mediante el literal #{ }
* las listas, en donde los elementos tienen un orden y puede haber elementos repetidos. Se definen mediante el literal [ ].

# Testeo unitario automatizado

Wollok cuenta con una herramienta para definir test unitarios automatizados, que se inscribe en la tendencia actual de los lenguajes de programación de uso profesional de utilizar el concepto Unit Test como parte del proceso de desarrollo de software. Un test, esencialmente, es una porción de código Wollok en la que se describe una determinada situación, mediante el envío de mensajes a los objetos correspondientes, y se especifica cuál es el resultado esperado.

## Unitario

El tipo de testeo que abordamos se califica como "unitario", porque la estrategia, en vez de hacer una gran y extensa prueba de un programa completo, se basa en identificar unidades significativas del código y probar casos puntuales donde éstas intervengan. Vamos a definir tests para los casos de prueba, explicitando mediante código la vinculación entre las situaciones planteadas y los resultados esperados.

Si mediante un mismo test probamos muchas cosas a la vez, cuando el test falla es más difícil saber cuál de los múltiples motivos falló. En cambio, si en un test probamos una sola cosa, al encontrarnos con que falla, tenemos una mayor certeza sobre cuál es el problema a corregir. Por este motivo, es fundamental el carácter unitario del test.

Los mismos principios de modularización y delegación de responsabilidades que caracteriza al paradigma de objetos, ayudan a que sea relativamente sencillo identificar las unidades sobre las que se van a definir los test.

Por otra parte, la forma en que anteriormente planteamos los casos de prueba ya tuvo en cuenta implícitamente los criterios de unicidad, por lo que -entendidos de esta manera- a cada caso de prueba le va a corresponder un test específico.

## Automático

La característica fundamental de los tests es que se puede automatizar su ejecución y consecuentemente su validación.

Como en el código de cada test, junto a la descripción de la situación a probar se especifica el resultado esperado, la validación es realizada por la misma máquina, sin mediar la interpretación de la persona, logrando mayor velocidad y confiabilidad.

A su vez, tratándose de código escrito, los tests quedan guardados al igual que el código de la solución propiamente dicha y cuando se desea, se pueden ejecutar todos de una vez, obteniendo un informe del resultado de cada uno de ellos. De esta manera, cuando en el proceso de desarrollo, al correr los tests se detecta algún problema y se corrige la solución, es sencillo volver a ejecutar todas las pruebas anteriores. También, si una solución que ya se probó que funciona correctamente se quiere refactorizar, basta con correr nuevamente todos los tests para garantizar que sigue funcionando adecuadamente.

Ante el aumento en la cantidad y variedad de tests, éstos se pueden agrupar y organizar de diferentes maneras, para permitir variantes en su ejecución.

## Independencia

Una característica que se desprende de su carácter unitario y que posibilita la automatización de los tests sin generar conflictos entre ellos, es que cada uno se concibe en forma independiente de cualquier otro test.

La lógica de ejecución de tests parte del supuesto que cada uno se corre a partir de la situación inicial del sistema, es decir que el ambiente se reinicia entre test y test, garantizando su total independencia.

Además, ante la detección de un problema por parte de algún test, los demás pueden seguir ejecutándose sin inconvenientes. El informe final detalla cuáles tests ejecutaron sin inconvenientes y cuáles detectaron algún problema.

## Información sintética y precisa

Teniendo en cuenta que la importancia de una prueba está dada por su capacidad para detectar problemas, la información más valiosa que aporta no es cuando funciona, sino cuando nos permite darnos cuenta que el resultado obtenido no es el esperado. En este caso, es importante que nos oriente con precisión acerca de dónde se produjo el problema y nos pueda brindar la mayor información posible.

Ya sea que se decida ejecutar uno, muchos o todos los tests, cuando el test corre bien, la información que arroja es bien sintética, simplemente dice que funcionó: no hay nada más que agregar que nos distraiga. En cambio, cuando un test falla, nos advierte con mayor contundencia y detalla cuál era el resultado esperado y el que realmente encontró.

## Definición de tests

El elemento clave es el objeto assert, un WKO que viene ya definido en el lenguaje, que entiende los mensajes que se utilizan en el siguiente ejemplo (y otros más que oportunamente presentaremos) y que permite una forma de plantear las expectativas declarativamente, delegando internamente al framework de testing que provee Wollok toda la lógica de la ejecución y validación automatizada.

### Describe

Identificamos a un conjunto de tests con la palabra reservada describe y un nombre expresivo con el cual lo identificamos. Al igual que los objetos y las clases, se utilizan { } para delimitar el inicio y fin de de la presente entidad, agrupando a todos los tests que forman parte de ella.

### Test

Dentro del describe, cada test se define de la siguiente manera:

* Cada uno comienza con la palabra reservada test seguido de una cadena de caracteres que explique lo que se está probando. Es importante hacer una buena descripción porque esa misma leyenda es la que va a aparecer en el informe de errores, y cuando el test falle nos va a permitir detectar más fácilmente cuál fue el problema.
* Se usan {} para marcar el inicio y fin del test, de igual manera que los métodos.
* En las primeras líneas, salvo tests muy sencillos, generalmente es necesario enviar los mensajes que van configurando la situación a probar. Es lo que en los casos de pruebas denominamos "acciones".
* Por último, se le envía un mensaje al objeto assert para realizar concretamente la prueba, contrastando lo que sé que se quiere probar con lo que se espera. Es lo que en los casos de prueba titulamos "validación". Hay básicamente tres alternativas:
  + El mensaje típico es equals, con dos parámetros: en primer lugar, va el valor que se espera que retorne el mensaje que va en segundo lugar. Se lo puede ver en la mayor parte de los tests.
  + En los casos donde se espera que la respuesta sea true, se le envía el mensaje that, con un único parámetro que es el mensaje booleano que se espera que se valide afirmativamente. Se lo puede ver en el segundo test.
  + En forma análoga, existe un mensaje notThat que testea que no se verifique el mensaje booleano. Se lo puede ver en el último de los tests.

# Propiedades en Wollok

Una propiedad en Wollok permite explicitar un contrato específico para una referencia:

* si la referencia es variable (var), Wollok genera su getter y setter
* si la referencia es constante (const), Wollok genera el getter

La creación de los accessors es implícita, no se visualiza en el código ya que la idea es conservar únicamente el comportamiento definido por el negocio.

# Clase

Entre los muchos objetos que pueden formar parte de un sistema, hay sin duda algunos muy diferentes a otros, con responsabilidades distintas, pero también muchos objetos son similares entre sí en los atributos que posee y en

los mensajes que son capaces de responder. Esto motiva la necesidad de “clasificar” los objetos, agrupándolos en función de sus semejanzas.

En concreto, los objetos que responden a los mismos mensajes de igual manera y que tienen una estructura interna igual (las mismas variables, con diferente valor para cada objeto), se clasifican juntos en una determinada “clase”.

La clase describe completa y detalladamente la estructura de información y el comportamiento que tendrá todo objeto de esa clase, o sea, define el conjunto de variables de instancia y de métodos que determinan cómo van a ser y cómo se van a comportar sus objetos. En otras palabras, determina cómo es y cómo actúa cada objeto.

La clase es lo genérico: es el patrón o modelo (“plano”, “molde”) para crear objetos. Cada objeto tiene su propia identidad, con una posición de memoria independiente de los otros objetos de la misma clase donde se almacenan sus valores para cada una de sus variables.

Se dice que todo objeto es una “instancia” de una clase, porque es creado a partir de ella, es “instanciado”. No existen objetos que no sean instancia de alguna clase y todo objeto conoce de qué clase es instancia.

La clasificación de un objeto se determina en función de los mensajes que se le van a enviar a él los restantes objetos. Para clasificar es necesario tener en cuenta que un objeto pertenece siempre a la misma clase: no puede dejar de ser

de una clase y pasar a ser de otra.

## 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.

Pero además de la clase, nosotros podemos definir otros tipos para el mismo objeto. Por ejemplo, podemos definir que un televisor, 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).

# 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.

## 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

}

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

## 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.

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:

## Parámetros nombrados

Wollok permite nombrar las referencias al construir el objeto, lo que ayuda a darle más claridad a la instanciación.

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

}

}

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

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 quién 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.

# Errores y excepciones

Una excepción es un evento que altera el flujo normal de mensajes entre objetos. Las excepciones tienen orígenes diferentes: desde errores serios de hardware, como la ruptura del disco rígido, hasta errores simples de programación, como tratar de acceder a un elemento fuera de los límites de un vector, dividir por cero o enviar un mensaje a una referencia que no apunta a ningún objeto.

## Alternativas para modelar una excepción

### Códigos de error

Una opción más interesante podría ser utilizar códigos de retorno numéricos:

method poner(cantidad) {

if (cantidad > 0) {

plata += cantidad

} else {

return -1

}

}

Entonces el método poner ya no solo tiene efecto colateral, sino un valor de retorno que hay que inspeccionar a su salida.

Esta estrategia tiene algunas desventajas

* ahora la definición del método no es consistente, ¿qué pasa si no hay error? ¿qué devuelve?
* obliga a quien llama a preguntar el valor de retorno antes de continuar con la siguiente línea de una secuencia de instrucciones,
* cuando hay varios mensajes de error posibles las preguntas se encadenan en ifs que dificultan el seguimiento
* los números mágicos -1, -2, no tienen descripciones representativas. Por eso a veces se reemplazan los números con mensajes de error alfanuméricos, aunque sigue siendo difícil rastrear el origen del problema.

## Excepciones

La tercera opción es lanzar una excepción por cada condición que salga del flujo normal, en particular

1. al sacar o poner una cantidad menor a cero
2. al sacar más de lo que permita el monedero

### Cómo generar una excepción

En el método poner tenemos que contemplar que la cantidad sea positiva:

method poner(cantidad) {

if (cantidad < 0) {

throw new UserException("La cantidad debe ser positiva")

}

plata += cantidad

}

¿Cómo funciona? Si la cantidad es menor a cero, se lanza una excepción cortando la secuencia de envío de mensajes que tenga el método: nunca llegarán a incrementar la cantidad de plata al monedero. Entonces quien envió el mensaje poner() al objeto monedero recibirá la excepción.

La clase UserException la definimos con un constructor que recibe el mensaje como parámetro: class UserException inherits Exception { }

## Excepciones de usuario y de programa

En base al cliente que va a recibir la excepción, podemos clasificar a las excepciones en dos tipos diferentes:

* Excepciones de usuario o de dominio: ocurren en el uso de la aplicación y son entendibles para el usuario final (“no hay saldo en la cuenta corriente”, “no hay stock del producto a facturar”, “no hay precio del producto a facturar”, etc.)
* Excepciones de programa: se producen cuando se ejecuta código de la aplicación y las puede analizar un especialista técnico (“falló el acceso al motor de la base de datos”, “hubo división por cero”, “el objeto no entiende este mensaje - MessageNotUnderstoodException”, etc.)

La naturaleza de ambos tipos de excepción son diferentes: en general las excepciones de negocio (o de aplicación) requieren que el usuario corrija la información que quiere ingresar al sistema (y valorice el producto, o bien seleccione un producto alternativo para facturar, o trate de sacar plata de otra cuenta bancaria), en tanto que las excepciones de programa requieren una corrección por parte de un usuario técnico (que chequeará la conexión a la base de datos o bien corregirá el código que originó el error).

Por lo tanto, las acciones a tomar cuando armamos cada tipo de excepción son diferentes: en las excepciones de negocio intentamos que el usuario vea una pantalla amigable donde le mostramos el problema que hubo al tratar de completar una acción con un mensaje representativo (e incluso proponiéndole soluciones alternativas para que la tarea se realice), mientras que en las excepciones de programa también mostramos una pantalla amigable al usuario, pero reservamos todos los detalles internos al desarrollador. De esa manera, las excepciones terminan siendo una herramienta más que ayuda a que nuestra aplicación se vuelva más robusta y confiable.

Entonces, para el componente de UI (interfaz de usuario), quizás sea más conveniente no tener que atrapar distintas excepciones, sino diferenciar las que son de negocio y las que no. En el ejemplo del monedero, si tenemos una tecnología de interfaz de usuario similar a la que venimos trabajando en el dominio, podremos hacer algo parecido a ésto:

method sacar() {

try {

monedero.sacar(<<monto>>)

} catch e : UserException {

self.mostrarMensajeDeAdvertencia(e.getMessage())

} catch e : Exception {

e.printStackTrace()

self.mostrarMensajeDeError("Ocurrió un error en la aplicación. Consulte al administrador.")

}

}

# Test Unitarios Automáticos Avanzados

## Inicialización del describe

A continuación de la especificación del describe con su correspondiente nombre, y previo a la enumeración de los tests que abarca, se pueden declarar variables y constantes. Al igual que cuando se definen en objetos o clases, pueden inicializarse ya sea con valores puntuales, instanciando clases existentes o con el valor de retorno de cualquier mensaje. El alcance de estas variables y constantes son todos los tests del describe (y tambíen fixture y métodos). Se permite definir constantes sin un valor inicial, siempre que sean inicializadas en el fixture.

Importante: Tener en cuenta que el estado de estas variables (y constantes) se reinicia con el valor especificado previo a ejecutar cada uno de los tests, al igual que todos los objetos existentes en el ambiente.

## Fixtures

Cuando se requiere realizar previo a cada test otras acciones de configuración de la situación inicial más complejas, para las cuales no es suficiente la inicialización de variables, se puede definir un fixture. Se lo hace precisamente con la palabra reservada fixture, sin agregar identificador y luego entre { } se detalla la secuencia de mensajes que se necesite, de manera similar a los constructores de las clases. Se especifica luego de la declaración e inicialización de variables y previo a los tests.

## Métodos auxiliares

Para ciertas acciones que se repiten entre algunos tests pero no son comunes a todos, se pueden definir métodos auxiliares, de igual manera que los métodos de cualquier objeto. Su nombre es cualquier identificador válido, puede tener parámetros, retornar valores, y ser invocado desde cualquier lugar del describe, -un test, otro método, la sección de inicialización-, utilizando self.

## Tests completos

Considerando todos los elementos mencionados, el orden de ejecución de cada uno de los tests del describe será:

* Reseteo del ambiente
* Inicialización de las variables de la descripción
* Fixture
* Test propiamente dicho

## Code coverage

El code coverage es una métrica utilizada para medir qué nivel de cobertura de código tienen nuestros tests. Es un número que va de 0 a 100, donde 1 es muy poco y 100 es demasiado. ¿Por qué demasiado? Porque a medida que nos acercamos al 100% nuestros tests pierden valor (por ejemplo, porque nos obsesionamos con probar getters, setters, métodos con lógica trivial, etc.) Además, conforme aparecen nuevos métodos de negocio no siempre podemos contemplar y generar los casos de prueba para todos ellos.

No hay un número mágico, es una cuestión de equilibrio y razonabilidad en función de las características del desarrollo y del equipo que lo lleva adelante. Si bien se busca que sea un porcentaje elevado, llega un punto donde dedicar demasiado esfuerzo a incrementar levemente el porcentaje de cobertura se vuelve contraproducente.

# Colecciones revisited

Hasta ahora definíamos colecciones de dos maneras, utilizando literales:

* #{} definía un conjunto, en el cual no había orden ni tampoco elementos repetidos
* [] definía una lista en el cual los elementos conservaban el mismo orden en el que fueron ingresados y el mismo elemento podía estar más de una vez

Bien, ahora que conocemos el concepto de clase como un mecanismo de agrupación de objetos, presentamos dos tipos de colecciones de Wollok: las clases Set y List. La primera define el conjunto matemático (sin orden ni duplicados) y la segunda la lista (cuyos elementos respetan el orden de ingreso y donde puede haber repetidos).

**SORTED BY + FOR EACH**

**Tipos de colecciones Set vs. List**

# Herencia

**SUPERCLASE. SUPER. SELF.**

## Clases abstractas y concretas

**DEFINICION ABSTRACTA Y CONCRETA**

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.

## 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.

**Instancias o clases**

# Herencia de Constructores

El mecanismo de herencia permite que una clase tome definiciones de su superclase, que por el momento eran atributos y comportamiento. Los constructores también participan de este concepto.

## La definición de constructores inhabilita la herencia

class Pelota {

var color = "blanco"

constructor() { }

constructor(\_color) { color = \_color }

}

class PelotaFutbol inherits Pelota {

var duenio

constructor(\_color, \_duenio) {

color = \_color

duenio = \_duenio

}

}

Como consecuencia de esa definición Wollok inhibe los constructores que la jerarquía de superclases tenga. Por lo tanto, la única forma correcta de construir una pelota de fútbol es pasándole el color y el dueño.

## Delegación de constructores con super

Si recibiéramos el color y en función de ese color determináramos el precio, o si estuvieran involucradas más tareas, sería bueno no repetir esa línea en la superclase y en todas las subclases que lo necesiten. Para tal fin vamos a delegar el constructor al constructor de la superclase, de la siguiente manera:

class Pelota {

var color = "blanco"

constructor() { }

constructor(\_color) { color = \_color }

}

class PelotaFutbol inherits Pelota {

var duenio

constructor(\_color, \_duenio) = super(\_color) {

duenio = \_duenio

}

}

# 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.

## 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.

## 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

# Igualdad e identidad

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

## Identidad

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

## 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.

# Elementos de Diseño

## Componentes

¿De qué hablamos cuando hablamos de componente? Existen varios puntos de vista:

* un componente puede ser un objeto, o una clase
* puede ser un conjunto de objetos/clases que cumplen un objetivo. Ej: un alumno con sus notas modela la carrera académica de un alumno.
* un componente puede ser un módulo o un sistema que se comunica con otro módulo o sistema. Ej: el módulo de seguimiento de carrera académica y el módulo de inscripciones a los cursos.

Cuando decimos que un componente tiene responsabilidades, ¿cómo se marcan esas responsabilidades? A partir de los métodos que implementa.

## Cohesión

* Una clase es cohesiva si podemos definirle un objetivo claro y puntual.
* Un método es cohesivo si tiene un único objetivo.

## Acoplamiento

Es el grado en que los componentes de un sistema se conocen.