Paradigma
Orientado a Objetos

Módulo 04: Objetos básicos.
Números. Strings.
Booleanos. Fechas.
Bloques. Colecciones.

por Fernando Dodino

Versión 3.1
Agosto 2019

Indice

La distribución de Wollok viene con objetos básicos que nos resultarán muy útiles para poder construir nuestras propias definiciones.

1 Números

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

- un número no cambia su estado interno
- la suma de 1 + 2 resulta en un nuevo número que representa al 3.

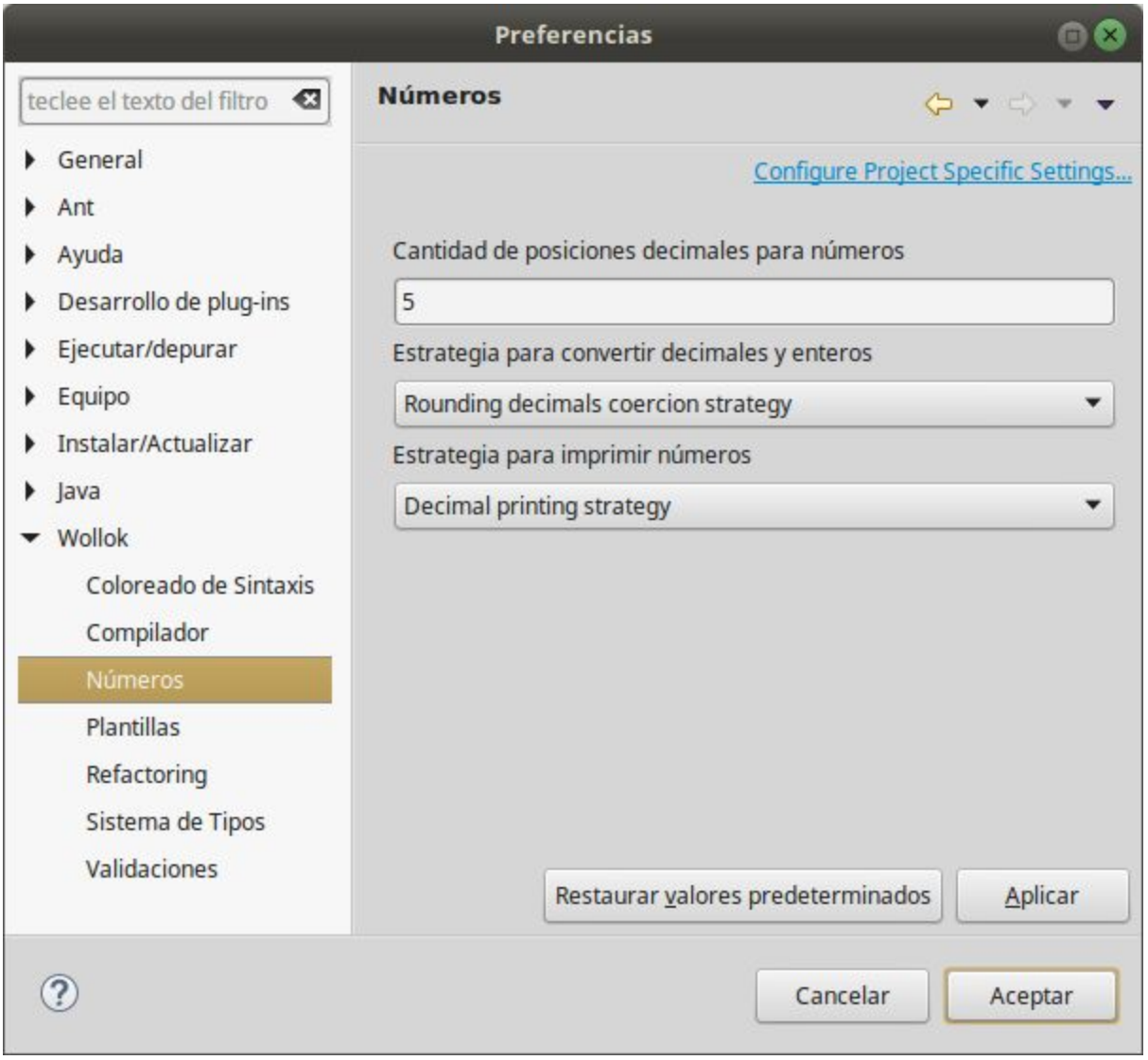
```
const a = 1
var b = a + 10  // suma
b = b - 1       // resta
b = b * 2       // multiplicación
b = b / 2       // división
b = b % 2       // resto
b = b ** 3      // elevado a (3 en este caso)
5.between(2, 7) // preguntamos si 5 está entre 2 y 7 ==> sí
3.min(6)        // el menor número entre 3 y 6 ==> 3
3.max(6)        // el mayor número entre 3 y 6 ==> 6
(3.1416).truncate(0)  // la parte entera de 3.1416 ==> 3 --
(3.1416).truncate(2)  // 3.14
(3.1416).roundUp(0)   // el primer entero mayor a 3.1416 ==> 4  --
(3.1416).roundUp(2)   // 3.15
```

1.1 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

La configuración de números se activa mediante el menú Ventana > Preferencias, en la solapa Wollok > Números:



Para una configuración por defecto, con cinco decimales como máximo, redondeando para arriba y con la impresión de los decimales representativos, vemos algunas operaciones:

```
>>> 5.2912413
5.29124
>>> 5.2912488
5.29125
>>> 1.000005 - 1.000004
0.00001
>>> 1.000002 - 1.000001
0
```

Si configuramos que en lugar de redondear impida el trabajo con más de 5 decimales, la respuesta por consola será diferente:

```
>>> 5.2912413
```

```
wollok.lang.Exception: El valor 5.2912413 debe tener menos de 5 posiciones decimales
```

2 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)`

no modifica el objeto original.

```
const hecho = true and true
const esTrue = true
const esFalse = false

const seraFalse = esTrue and esFalse

const seraTrue = esTrue or esFalse

const seraTrue = not false
```

Para aquellos que estén acostumbrados a los operadores con símbolos (en lenguajes como C o java) pueden usar esa sintaxis si se sienten más cómodos:

- and: a `&&` b
- or: a `||` b
- not: `!`a

3 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”).

```
const holaMundo = unString + otroString + " !"
```

4 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(day = 24, month = 11, year = 2017)
        // se ingresa en formato día, mes y año
```

Algunas operaciones que podemos hacer con las fechas son:

```
>>> const hoy = new Date()
>>> hoy
24/11/2017
>>> hoy.plusYears(1)    // sumo un año
24/11/2018              // devuelve una nueva fecha
>>> hoy.plusMonths(2)   // sumo 2 meses
24/1/2018
>>> hoy.plusDays(20)
14/12/2017
>>> hoy.isLeapYear()    // pregunto si el año es bisiesto
false
>>> hoy.dayOfWeek()     // qué día de la semana es
friday[]
>>> hoy.month()
11
>>> hoy.year()
2017
>>> const ayer = hoy.minusDays(1)
                        // resto un día para obtener el día de ayer
>>> ayer < hoy          // comparo fechas
true
>>> ayer - hoy          // comparo fechas
-1                      // diferencia en días entre ayer y hoy
>>> const haceUnMes = hoy.minusMonths(1)
>>> ayer.between(haceUnMes, hoy)
true                    // ayer está entre hace un mes y hoy
```

5 Lambdas: objetos bloque

5.1 Introducción, ejemplo básico

Al igual que en otras tecnologías, Wollok nos permite tener un objeto que representa un bloque de código[1], de manera de

[1] A veces se referencia este tipo de expresiones como *closure* (sobre todo quienes vienen del mundo Java). En términos generales esto no es correcto, dado que son ideas diferentes: una lambda es una función/método que no tiene nombre, mientras una closure o cierre o clausura es un método que puede acceder a las variables disponibles en el contexto en que se la declaró

- 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,
    - ante un evento
    - o bien lo ejecutamos *n* veces hasta que se cumpla una condición: por ejemplo para reprocesar una instrucción hasta que no haya errores.

Veamos el primer ejemplo básico, un bloque de código que permite conocer el valor absoluto del número 4:

```
>>> const abs4 = { => 4.abs() }
>>> abs4
{ => 4.abs() }

>>> abs4.apply()
4
```

En la primera línea creamos la expresión lambda. En la última línea evaluamos el código que contiene abs4, y devolvemos el valor 4.

5.2 Bloques con un parámetro

Vamos a mejorar el ejemplo anterior, pasando como parámetro un número cuyo valor absoluto queremos conocer:

```
>>> const abs = { numero => numero.abs() }
>>> abs
{ numero => numero.abs() }
>>> abs.apply(-8)
8
```

La operatoria es exactamente igual, sólo que como vemos el método apply está sobrecargado para aceptar un parámetro.

5.3 Bloques con dos parámetros

También podemos definir lambdas que reciben 2 parámetros, por ejemplo para seleccionar el mayor número entre dos, o bien para sumarlos:

```
>>> { num1, num2 => num1.max(num2) }.apply(4, 2)
```

(esto incluye variables que no son locales dentro del bloque que definimos, tiene un alcance mayor).

```
4
>>> { num1, num2 => num1 + num2 }.apply(4, 2)
6
```

5.4 Contexto de los bloques

Un dato importante de los closures es que no solo acceden a sus parámetros, sino también a cualquier otra referencia en el contexto donde fueron definidas. Esto las vuelve realmente poderosas. Veamos un ejemplo muy sencillo:

```
var to = "world"
const helloWorld = { "hello " + to }
helloWorld.apply() == "hello world"              // true

to = "someone else"
helloWorld.apply() == "hello someone else"       // true
```

Se podrá ver que el closure accede a la variable “to” que es definida fuera del contexto del closure mismo, dentro del programa. Si cambiamos esta referencia, este efecto se propaga al closure (como se muestra en la segunda llamada, el valor devuelto es diferente).

5.5 Comparación con el paradigma funcional

`( \num1 num2 -> num1 + num2 )` Haskell	`{ num1, num2 => num1 + num2 }` Wollok

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 **declarativid**. 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).

6 Introducción a colecciones

6.1 ¿Qué es una colección?

La colección nos permite representar un conjunto de objetos relacionados:
los jugadores de un equipo de fútbol, un cardumen, una lista de cosas a comprar en el supermercado, las cosas que un héroe guarda en su mochila, un ejército, son ejemplos de este tipo de abstracciones.

Otra definición posible es que una colección nos sirve para modelar una relación 1 a N:

- Una factura tiene muchas líneas con productos
- Un escritor publicó varios libros
- Una fiesta tiene muchos invitados
- Un héroe tiene que cumplir varias misiones

A primera vista una colección es un conjunto de objetos. Si la vemos con más detalle 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.

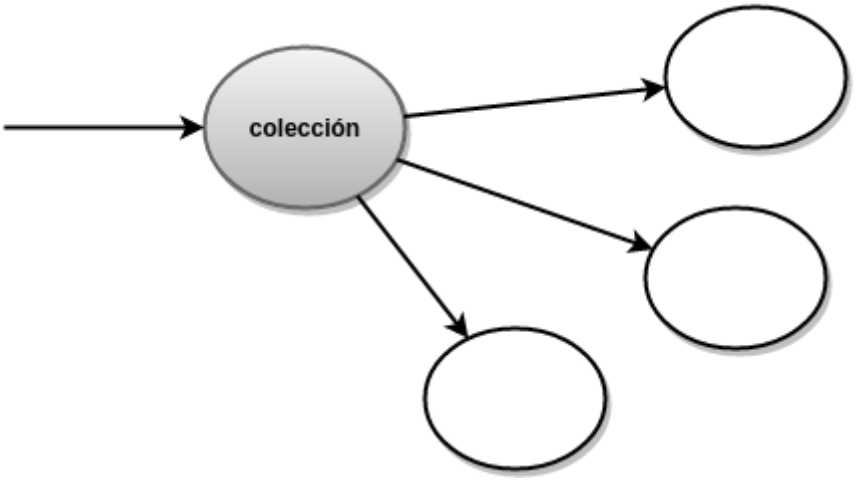


Figura 3.1: La colección es un conjunto de referencias a otros objetos

6.2 Representación

Podemos graficar la relación dinámica entre un equipo de fútbol y los jugadores que lo integran mediante un diagrama de objetos. Este es un diagrama con características *dinámicas*, porque muestra el estado de los objetos en un momento determinado.

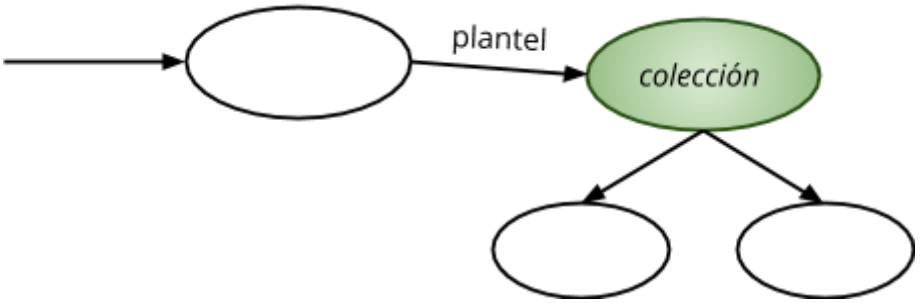


Figura 3.2: El plantel de jugadores de un equipo como una colección de objetos

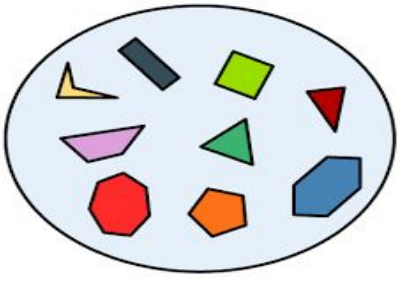
6.3 Interfaz de una colección

Supongamos que tenemos un álbum de fotos, otra representación posible de una colección de objetos. ¿Qué podemos hacer con esas fotografías?

- Mirarlas, "recorrerlas": iterar una colección
- Averiguar cuántas fotos hay: saber el tamaño de una colección
- Saber si está una determinada foto en el álbum, es decir, si un elemento pertenece a la colección
- Pegar una foto nueva: agregar un elemento a la colección
- Regalar una foto a alguien: eliminar un elemento de la colección
- Seleccionar las fotos del viaje de 1999 a Ushuaia: filtrar elementos de una colección
- Saber las fechas en la que saqué mis fotos: transformar los elementos de una colección
- Saber si hay alguna foto de Navidad: determinar si algún elemento satisface un criterio

6.4 Intro a 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 #{ }

```
const numeros = #{1, 2, 3, 4}
```

las **listas**, en donde los elementos tienen un orden y puede haber elementos repetidos. Se definen mediante el literal [ ]

```
const fila = [francisco, mirta, hector]
```

7 Ejemplo integrador: Misiones de un héroe

En un sistema que administra diferentes tipos de héroe, a un héroe se le puede asignar misiones y nuevos objetivos. Por ejemplo, un mago puede encargarle buscar un ítem mágico en una montaña lejana. Un anciano puede encargarle liberar a su hija de los terribles trolls que habitan en la gruta de los sin nariz[2].

Cada vez que un héroe tiene uno de estos encuentros, anota los datos de la misión en su diario personal. Toda misión suma en el camino del héroe: las misiones tienen una recompensa de respeto, que luego puede canjearse por oro.

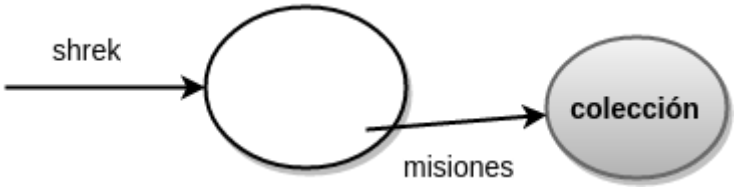
¿Qué abstracciones surgen? El héroe tiene una colección de misiones, algunas de las cuales puede ser

1. buscar un ítem mágico, que es difícil si el ítem a buscar queda a más de 100 kms. (la localización del ítem puede ir variando)
2. liberar a fiona, custodiada por una cantidad variable de trolls (si son 4 ó 5 la misión es difícil).

7.1 Shrek, nuestro único héroe en este lío

Definiremos un objeto shrek, un gran (anti)héroe. Por el momento sabemos que tiene misiones, la primera pregunta que nos tenemos que hacer es: ¿necesitamos mantener el orden de cada una de las misiones? ¿podemos asignarle dos veces la misma misión? A primera vista, el orden no parece importante y vamos a suponer que, de la misma manera que uno no puede bañarse dos veces en el mismo río[3], uno no libera a la misma doncella dos veces.

```
object shrek {
	const misiones = #{}
}
```



[2] El ejemplo completo puede descargarse de https://github.com/wollok/heroes-con-objetos

[3] En realidad el dicho de Heráclito es *ποταμοῖς τοῖς αὐτοῖς ἐμβαίνομεν τε καὶ οὐκ ἐμβαίνομεν, εἶμεν τε καὶ οὐκ εἶμεν τε.*
En los mismos ríos entramos y no entramos, [pues] somos y no somos [los mismos].
Esto implica que tanto el río como el que se baña en él cambian todo el tiempo.

Fig.3.3: Misiones es una referencia a un *conjunto*

La referencia es constante, porque agregaremos o eliminaremos elementos pero siempre apuntando a la misma colección.

7.2 Primera misión: liberar a Fiona

Shrek tiene su primera misión asignada por Lord Farquaad: liberar a una doncella encerrada en la torre más alta de un castillo protegido por un ejército de 5 trolls[4]. Creamos un objeto que representa la misión de liberar a Fiona

```
object liberarAFiona {

}
```

El lector observará que no definimos todavía comportamiento ni atributos: no queremos adelantarnos ni tomar decisiones antes de tiempo. Por el momento nuestro objetivo es incorporar la misión al héroe y para eso necesitamos mandarle un mensaje a shrek:

```
>>> shrek.agregarMision(liberarAFiona)
```

Entonces en shrek, debemos escribir el método agregarMision(), que delega la acción al objeto que está representando la colección:

```
object shrek {
	const misiones = #{}
	method agregarMision(_mision) { misiones.add(_mision) }
}
```

Cuando enviemos el mensaje a shrek

```
>>> shrek.agregarMision(liberarAFiona)
```

en nuestro ambiente tendremos estas referencias:

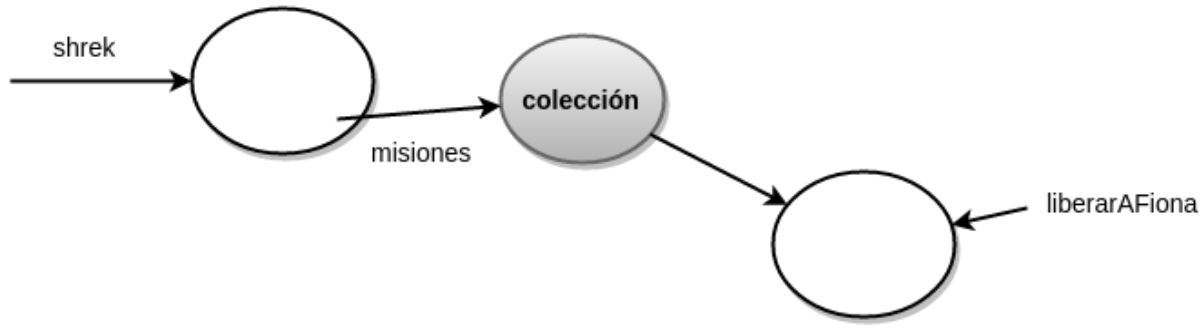


Fig 3.4: Primera misión para shrek

[4] Luego de unos años, la Dragona decidió tercerizar la seguridad.

7.3 Cuántas misiones tiene un héroe

¿Cómo sabemos cuántas misiones tiene un héroe? Le preguntamos a shrek
`shrek.cantidadDeMisiones()`

Son dos momentos diferenciados:

1) primero la interfaz,
2) luego cómo implementarlo

```
object shrek {
	...
	method cantidadDeMisiones() = misiones.size()[5]
}
```

7.4 Segunda misión: buscar la piedra filosofal

Shrek se cruza inesperadamente con el director de una famosa escuela de magia, quien le pide que en secreto busque una famosa piedra filosofal que dejó olvidado una noche de borrachera en los montes Urales, a 40 kms. del pantano donde vive nuestro héroe.

Creamos entonces nuestro objeto buscarPiedraFilosofal:

```
object buscarPiedraFilosofal { }
```

Y corremos nuevamente la consola interactiva, agregando ambas misiones:

```
>>> shrek.agregarMision(liberarAFiona)
>>> shrek.agregarMision(buscarPiedraFilosofal)
```

Podemos volver a preguntar a shrek la cantidad de misiones, no esperamos sorprender al lector con la respuesta.

7.5 Filtrar misiones difíciles

Nuestro nuevo requerimiento necesita aplicar varios conceptos, vamos con el primero: las misiones “liberar a Fiona” y “buscar la piedra filosofal” tienen que entender el mismo mensaje.

```
>>> liberarAFiona.esDificil()
>>> buscarPiedraFilosofal.esDificil()
```

[5] Otra forma de definir el método es con la definición del cuerpo del método y el return explícito
`method cantidadDeMisiones() { return misiones.size() }`
Wollok permite esta definición simplificada del método cuando abarca una sola línea y devuelve un valor. En caso de duda recomendamos seguir con la sintaxis original.

Entonces, necesitamos que sean **polimórficos**. ¿Para quién es esto? No para las misiones, sino para shrek, que es quien posteriormente los va a usar. Pero no nos apuremos, ahora sí tenemos que definir el comportamiento en las misiones

```
object liberarAFiona {
	var cantidadTrolls = 5
	method esDificil() = cantidadTrolls.between(4, 5)
}

object buscarPiedraFilosofal {
	var kilometrosDistancia = 40
	method esDificil() = kilometrosDistancia > 100
}
```

Fíjense que no necesitamos -por el momento- definir *accessors* (*getters* / *setters*) para los atributos de los objetos liberarAFiona y buscarPiedraFilosofal.

7.6 Filtrar elementos de una colección

¿Cómo pregunto qué misiones difíciles tiene Shrek?

```
>>> shrek.misionesDificiles()
#{liberarAFiona}
```

Las colecciones definen un mensaje filter, al que le podemos pasar el criterio que deben cumplir los elementos que nos interesan:

```
object shrek {
	...
	method misionesDificiles() =
		misiones.filter({ mision => mision.esDificil() })
}
```

Más adelante veremos en profundidad el objeto que le estamos pasando como parámetro a filter, por ahora nos alcanza con ver la similitud que tiene esta solución con la forma de trabajo en el paradigma funcional.

Para resolver las misiones difíciles

- shrek conoce a sus misiones
- pero delega la respuesta a cada misión, que entiende el mensaje `esDificil()`. No importa cómo lo implementan, lo importante es que todos devuelven un valor booleano que permite discriminar misiones fáciles de difíciles.

7.7 Filter y el efecto colateral

El resultado de enviar el mensaje misionesDifíciles() no altera las referencias de la colección original:

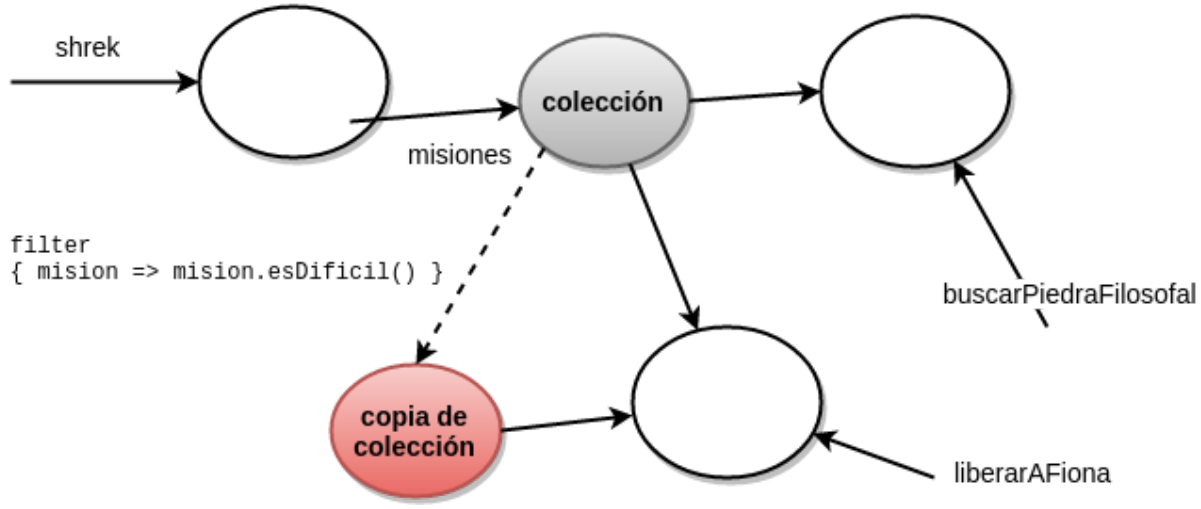


Figura 3.5: Filter devuelve una copia de la colección con los elementos que cumplen una condición

Esto permite hacer consultas sin que haya efecto sobre las referencias originales, no sería lógico que al querer conocer los clientes que nos deben plata perdamos los que sí pagaron[6].

7.8 Map: Encargados de una misión

Cada misión tiene un solicitante, en particular

- liberar a Fiona fue solicitado por Lord Farquaad
- mientras que buscar la piedra filosofal fue solicitado por Mr. Dumblecofcofdore

Una forma de modelar esto es mediante una referencia o variable, otra por el momento es tener un método que retorne una constante. No nos vamos a preocupar mucho más por el momento, más adelante tendremos la oportunidad de mejorar esta idea:

```
object liberarAFiona {
	var cantidadTrolls = 4
	method solicitante() = "Lord Farquaad"
	...
```

[6] De la misma manera funciona el filter que hemos visto en Haskell, facilitado por la ausencia de asignaciones destructivas

```
object buscarPiedraFilosofal {
	var kilometrosDistancia = 40
	method solicitante() = "Mr DumblecofcofDore"
	...
```

¿Cómo le pedimos a shrek los solicitantes de todas sus misiones?

```
>>> shrek.agregarMision(liberarAFiona)
>>> shrek.agregarMision(buscarPiedraFilosofal)
>>> shrek.solicitantesDeMisMisiones()
#{"Lord Farquaad", "Mr Dumblecofcofdore"}
```

Ahora podemos pensar en el método solicitantesDeMisMisiones() que tiene que aplicar una transformación a cada elemento de la colección:

INPUT: una misión	OUTPUT: su solicitante (un String)
cómo lo logramos: `mision.solicitante()`	

Nos ayudamos con el mensaje map() a la colección:

```
object shrek {
	...
	method solicitantesDeMisMisiones() =
		misiones.map({ mision => mision.solicitante() })
}
```

Al igual que filter, map no tiene efecto colateral:

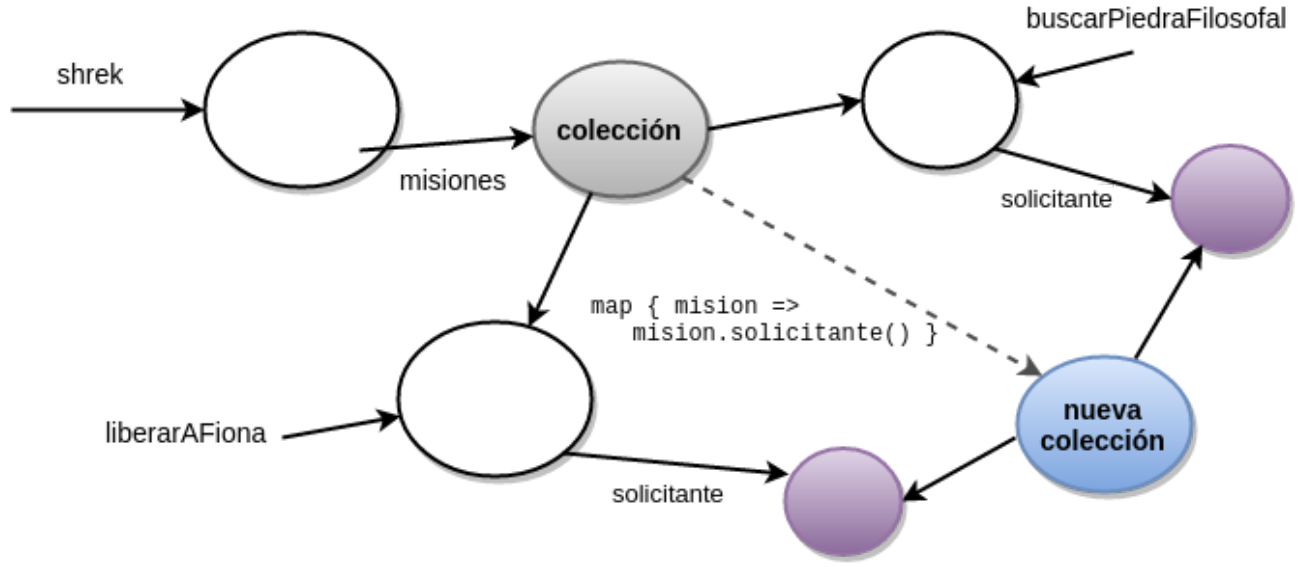


Figura 3.5: Map devuelve una colección nueva con los elementos transformados

Esto tiene sentido, porque además pasamos de un conjunto de misiones a un conjunto de solicitantes.

7.9 Sumar puntos de recompensa

Cada misión tiene una cantidad de puntos de recompensa, un valor numérico. Los puntos de recompensa por liberar a fiona son 2 * cantidad de trolls que la custodian. Respecto a buscar el ítem mágico, son 10 puntos si la distancia a recorrer es más de 50 kilómetros ó 5 en caso contrario.

```
object liberarAFiona {
	...
	method puntosRecompensa() = cantidadTrolls * 2
}

object buscarPiedraFilosofal {
	...
	method puntosRecompensa() =
		if (kilometrosDistancia > 50) 10 else 5
}
```

Nuevamente, ¡podemos enviar mensajes a liberarAFiona o buscarPiedraFilosofal en forma indistinta! Encontramos otro contexto en el cual ambos objetos son **polimórficos**. Y esto nos permite resolver nuestro tercer requerimiento: queremos saber los puntos de recompensa de las misiones que tiene encargado un héroe, esto se logra... enviando el siguiente mensaje

```
>>> shrek.totalPuntosDeRecompensa()
15
```

claro está.

Lo resolvemos aprovechando otra construcción conocida por nosotros: sum, que necesita el criterio para sumar números obtenidos en base a elementos de una colección.

```
object shrek {
	...
	method totalPuntosDeRecompensa() =
		misiones.sum({ mision => mision.puntosRecompensa() })
}
```

Y aquí vemos claramente que shrek *aprovecha el polimorfismo* para sumar los puntos de recompensa de las misiones, sin importarle cómo lo termina implementando cada una.

8 Resumen

En este capítulo presentamos a los objetos básicos de Wollok: los números, los strings, los booleanos, las fechas, los bloques y finalmente las colecciones como una forma de tener una referencia a múltiples objetos, y comenzamos a conocer su rica interfaz que nos permite resolver requerimientos sin necesidad de tener que bajar demasiado a detalle el algoritmo.