Paradigma
Orientado a Objetos

Módulo 12:
Colecciones con clases.
Bloques de código.

por Fernando Dodino
revisión por Lucas Spigariol
Versión 3.2
Julio 2019

Indice

1 Colecciones revisited
    1.1 Interfaz de una colección

2 Misiones de un héroe, parte 2
    2.1 Definiciones: de objetos a clases
    2.2 Diagrama de clases de la solución
    2.3 Referencias: algunas pruebas
    2.4 ¿Colecciones sin polimorfismo?

3 Mensajes de colección
    3.1 Repaso de filter
    3.2 Repaso de la interfaz de las colecciones
    3.3 Encontrar un elemento que satisfaga una condición
    3.4 Ordenar una colección
    3.5 For-each

4 Tipos de colecciones Set vs. List

5 Dictionary (BONUS)

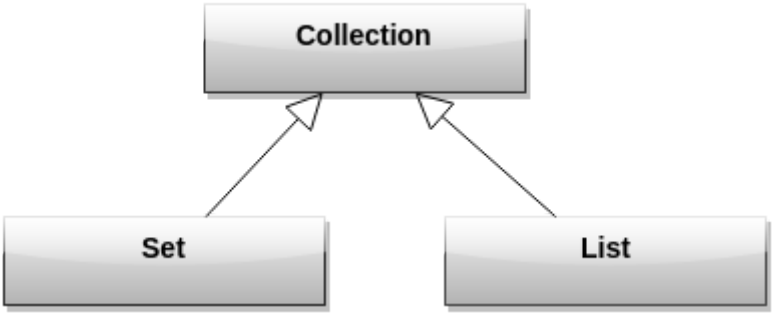
6 Resumen

1 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 la 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).



1.1 Interfaz de una colección

La interfaz de Set y List es la misma que conociste en el módulo 4, así que por este lado no hay cambios:

```
const unConjunto = new Set()
unConjunto.add(1)
unConjunto.add(3)
unConjunto.size()
```

2 Misiones de un héroe, parte 2

2.1 Definiciones: de objetos a clases

Para utilizar clases en nuestro ejercicio de los héroes, tenemos que reformular algunas cosas[1]:

- liberarAFiona es una instancia de LiberarDoncella
- buscarPiedraFilosofal es una instancia de BuscarItemMagico

[1] El ejemplo completo se puede descargar de https://github.com/wollok/heroes-con-clases/

- aprovecharemos las propiedades de Wollok para eliminar código *boilerplate*

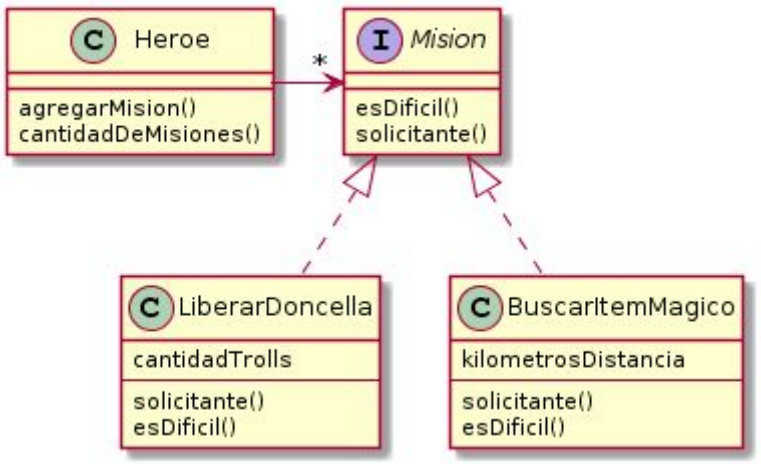
```
class LiberarDoncella {
	var property solicitante = ""
	const cantidadTrolls

	method esDificil() = cantidadTrolls.between(4, 5)

	method puntosRecompensa() = cantidadTrolls * 2
}

class BuscarItemMagico {
	var property solicitante = ""
	const property kilometrosDistancia

	method esDificil() = kilometrosDistancia > 100

	method puntosRecompensa() = if (kilometrosDistancia > 50) 10
else 5
}
```

El objeto shrek se transforma en una clase Héroe, con un atributo misiones que podría definirse con el literal #{}, o también construyendo explícitamente una instancia de Set:

```
class Heroe {
	var misiones = new Set()
	... el resto igual
```

2.2 Diagrama de clases de la solución

Realizamos un diagrama de clases de la solución generada:



Este es un diagrama con características *estáticas*, porque no depende de un caso particular sino que muestra las relaciones entre las clases.

¿Qué es lo que cuenta el diagrama?

- Que un héroe tiene **muchas** misiones, definido por el asterisco (*).
- ¿Qué significa la caja Misión? Una interfaz, ya que LiberarDoncella y BuscarItemMagico no comparten código, pero sí tienen en común que ambos implementan los mensajes solicitante() y esDificil()
- La línea punteada que termina en un triángulo cerrado marca una relación de **Realización** (implementa) entre LiberarDoncella/Misión y BuscarItemMagico/Misión
- La línea continua con una flecha abierta marca una relación de **Asociación** entre Héroe y Misión.
- Para más información recomendamos leer este apunte

La dirección marca qué objeto conoce a los otros: como la flecha va de Héroe a Misión sabemos que cada héroe tiene n misiones, no conocemos qué características tiene la relación de Misión a Héroe, pero se pueden dar dos opciones:

- una misión puede ser realizada por un héroe solo, en ese caso la relación Héroe-Misión es de uno a muchos
- una misión puede ser encarada por varios héroes. En ese caso la relación es de muchos a muchos.

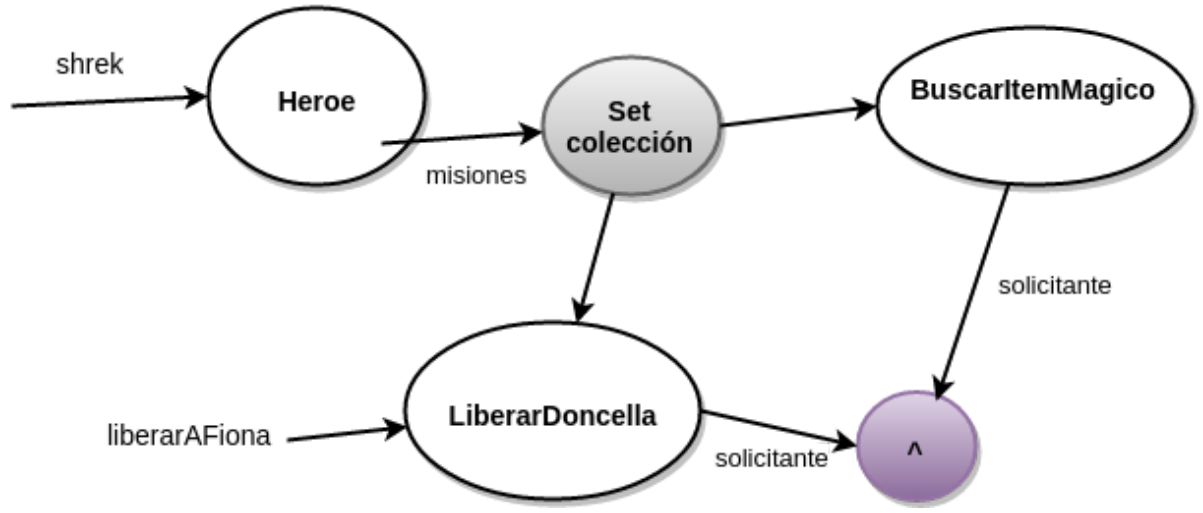
2.3 Referencias: algunas pruebas

Generamos las misiones de shrek

```
Wollok interactive console (type "quit" to quit):
>>> const shrek = new Heroe()
a Heroe[misiones=#{}]
```

```
>>> const liberarAFiona = new LiberarDoncella(cantidadTrolls = 5)
a LiberarDoncella[cantidadTrolls=5, solicitante=]
>>> shrek.agregarMision(liberarAFiona)
>>> shrek.agregarMision(new BuscarItemMagico(kilometrosDistancia =
40))
>>> shrek.cantidadDeMisiones()
2
```

Vemos el gráfico de los objetos en el ambiente (aquí vemos que la instancia de la clase LiberarDoncella tiene 2 referencias):



Este es un diagrama con características *dinámicas,* porque muestra el estado de los objetos en un momento determinado.

2.4 ¿Colecciones sin polimorfismo?

Si quiero cambiar la referencia de liberarAFiona

```
>>> liberarAFiona = 5
```

Eso produce un error

```
ERROR: No se pueden modificar las referencias constantes (linea: 1)
```

Esto es debido a que definimos nuestra referencia como constante (inmutable):

```
>>> const liberarAFiona = new LiberarDoncella(cantidadTrolls = 5)
```

¿Y si agregamos una referencia a un valor que no sea una misión?

```
>>> var liberarAPepe = 4
4
>>> shrek.agregarMision(liberarAPepe)
>>> shrek.cantidadDeMisiones()
3
```

Esto parece funcionar, el problema es cuando intente usar polimórficamente al entero con las misiones, es entendible que se produzca un error:

```
>>> shrek.solicitantesDeMisMisiones()
el mensaje de error es largo, la línea que nos marca el problema es ésta:
wollok.lang.MessageNotUnderstoodException: 4 no entiende el mensaje solicitante()
   at wollok.lang.Object.messageNotUnderstood(messageName,parameters) (classpath:/wollok/lang.wlk:228)
```

3 Mensajes de colección

3.1 Repaso de filter

Volvamos al método misionesDificiles, antes del objeto shrek y ahora de Heroe:

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

Sabemos lo que hace, filtra las misiones que son difíciles, lo importante es que Héroe delega el algoritmo de filtrado a la colección (el objeto que representa el conjunto de misiones). Y para eso, construye un **bloque de codigo** que recibe una misión y devuelve un valor booleano.

Lo podemos probar en el REPL:

```
>>> { mision => mision.esDificil() }.apply(new LiberarDoncella(cantidadTrolls = 4))
true
>>> { mision => mision.esDificil() }.apply(new LiberarDoncella(cantidadTrolls = 7))
false
```

Y podemos construir un filter *wollokiano*:

```
>> clase Set
      method filter(condicionFiltro) {
            const result = #{ }
            self.forEach({ elemento =>
                  if (condicionFiltro.apply(elemento)) {
                        result.add(elemento)
                  })
            }
            return result
```

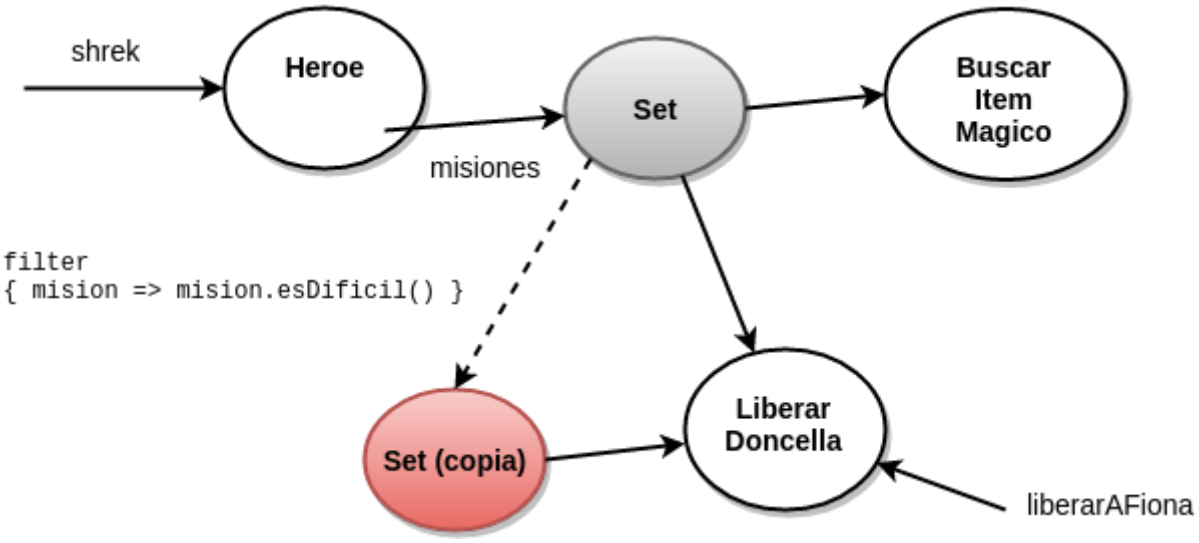
```
}
```

Claro, que al tenerlo resuelto, yo sólo me concentro en

- decir que quiero usar **filter**
- generar el **bloque de codigo** que defina el criterio de filtrado de los elementos

Así -como dijimos anteriormente- subimos el nivel de **declarativiad**.

Recordamos que el filter no tiene efecto colateral, se devuelve una nueva colección con los elementos que cumplen dicha condición:



3.2 Repaso de la interfaz de las colecciones

Recordemos que map permite transformar los elementos de una colección generando una nueva colección.

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

El método solicitantesDeMisMisiones utiliza una lambda que transforma una misión en un solicitante.

Para obtener el total de puntos de recompensa, utilizamos el mensaje sum:

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

Wollok también trae el mensaje fold, para reducir los elementos de una colección a un valor:

```
method totalPuntosDeRecompensa() =
      misiones.fold(0,
            { acum , mision => acum + mision.puntosRecompensa() })
```

Para el lector: puede probar en una consola REPL

```
>>> { acum , mision => acum + mision.puntosRecompensa() }
      .apply(0, new LiberarDoncella(cantidadTrolls = 5))
10

>>> { acum , mision => acum + mision.puntosRecompensa() }
      .apply(25, new LiberarDoncella(cantidadTrolls = 5))
35
```

3.3 Encontrar un elemento que satisfaga una condición

Si queremos saber qué misión tiene más de 5 puntos de recompensa, ¿cómo hacemos?

```
shrek.misionQueTengaMayorRecompensaQue(5)
```

Y lo implementamos así:

```
method misionQueTengaMayorRecompensaQue(puntos) =
   misiones.find({ mision => mision.puntosRecompensa() > puntos })
```

Una vez más utilizamos el concepto de **bloques de código** para devolver el primer elemento que cumple una condición[2].

3.4 Ordenar una colección

Para ordenar las misiones por puntos de recompensa:

```
method misionesOrdenadasPorPuntos() =
      misiones.sortedBy({ mision1, mision2 =>
            mision1.puntosRecompensa() < mision2.puntosRecompensa()})
```

Utilizamos un bloque, que define el criterio de ordenamiento para dos elementos y una condición

- si la condición se cumple, el primer elemento se ordena antes que el segundo
- en caso contrario, es el segundo elemento el que queda antes que el primero.

¿Cómo hacemos si queremos ordenar de mayor a menor? Por supuesto, tenemos que invertir la condición, para hacer que los elementos queden ordenados al revés que en la solución original:

[2] El lector atento podrá notar la presencia de una variable libre: el parámetro puntos. Y aquí comienza a introducirse la distinción entre closures y lambdas que mencionamos anteriormente. (recomendamos leer la respuesta del usuario silvali)

```
method misionesOrdenadasPorPuntos() =
      misiones.sortedBy({ mision1, mision2 =>
            mision1.puntosRecompensa() > mision2.puntosRecompensa()})
```

3.5 For-each

Al presentar el mensaje filter, estuvimos presentando “sin querer” el mensaje forEach, que aplica una operación sobre todos los elementos de una colección. Esto tiene sentido cuando esa operación tiene efecto colateral. Por ejemplo, si queremos modificar todos los solicitantes de las misiones que tiene un héroe.

Para resolverlo, enviamos un mensaje:

```
shrek.cambiarSolicitanteDeMisiones("señor X")
```

E implementamos el método cambiarSolicitanteDeMisiones en la clase Héroe:

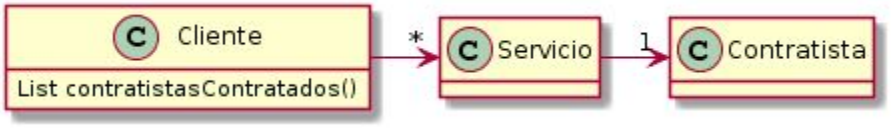
```
method cambiarSolicitanteDeMisiones(nuevoSolicitante) {
      misiones.forEach(
            { mision => mision.solicitante(nuevoSolicitante) })
}
```

- Hasta ahora, nos interesaba saber qué devolvía cada mensaje
- Al pedir shrek.cambiarSolicitante(...) lo que nos importa es que ese mensaje tenga efecto sobre shrek. Entonces para comprobar que el mensaje produjo ese efecto, lo tenemos que complementar con otro mensaje que muestre que las misiones tienen el nuevo solicitante. Una solución posible es generar el método getter para la colección de misiones.

Si queremos trabajar en forma declarativa, deberíamos utilizar forEach solo cuando necesitemos producir efecto sobre los elementos de una colección.

4 Tipos de colecciones Set vs. List

Consideremos el siguiente ejemplo:



Un cliente contrata servicios para arreglar su casa (pintura, plomería, albañilería, etc.). Cada servicio involucra diferentes contratistas. Tenemos un cliente Milena, y dos contratistas: Silvina y Eliana (modelados como instancias de la clase Contratista).

Cada vez que un cliente contrata a un contratista se genera un nuevo servicio donde se le asocia el contratista y la fecha de la contratación.

Un usuario nos reporta que este test falla, ¿qué podrá ser?

```
test "La lista de contratados por milena esta compuesta por silvina
y eliana" {
	const milena = ... creación de Milena ...
	milena.contratarA(silvina)
	milena.contratarA(silvina)
	milena.contratarA(eliana)
	milena.contratarA(eliana)
	assert.equals([silvina,eliana],
		milena.contratistasContratados().asSet())
}
```

Sabemos que contratistasContratados devuelve una lista. Como se que esa lista de contratistas va a devolver elementos duplicados, lo convierto a un conjunto enviando el mensaje asSet. Pero aun así, el test falla porque estoy comparando ese set contra... silvina y eliana que es... ¡**una lista!**

En Wollok

- dos conjuntos son iguales si tienen los mismos elementos **no importa el orden y no importa si para armar el conjunto repetimos varias veces los elementos**
- dos listas son iguales si tienen **los mismos elementos en el mismo orden** entonces ojo porque [1, 2, 3] es diferente de [1, 3, 2]

Algunas consultas para reforzar lo dicho:

```
>>> [1, 2, 3] == [1, 2, 3]
true
>>> [1, 2, 3] == [1, 3, 2]
false
>>> [1, 2, 3] == [1, 3, 2, 3]
false
```

Mientras que

```
>>> #{1, 2, 3} == #{1, 3, 2}
true
>>> #{1, 2, 3} == #{1, 3, 2, 3}
true
```

5 Dictionary (BONUS)

Tenemos el siguiente requerimiento: dada una colección de “Rendiciones de gastos”, donde cada rendición conoce

- la persona que realizó el gasto (por el momento es solo un String)
- la fecha en la que hizo el gasto
- la cantidad de plata que gastó

queremos poder saber los totales por mes y por persona.

Pensemos qué queremos: si tenemos estas rendiciones

Fecha	Quién	Cuánto
hoy	Juan	100
hace un mes	Juan	250
ayer	Juan	120
hoy	Ricky	500
anteayer	Ricky	50

```
>>> const reporter = new Reporter()
>>> reporter.mostrarTotalesPorCliente()
Juan - $ 470
Ricky - $ 550
>>> reporter.mostrarTotalesPorFecha()
a Date[day = 21, month = 9, year = 2016] - $ 250
a Date[day = 19, month = 10, year = 2016] - $ 50
a Date[day = 20, month = 10, year = 2016] - $ 120
a Date[day = 21, month = 10, year = 2016] - $ 600
```

Bien, para lograr eso, necesitamos la clase Rendición[3]:

```
class Rendicion {
	const property persona
	const property fecha
	const property total

	}
```

Nada del otro mundo. La clase Reporter tendrá en el constructor la generación del juego de datos:

[3] El ejemplo completo se puede descargar en https://github.com/wollok/dictionary-totales

```
class Reporter {
	const hoy = new Date()
	const rendiciones = [
		new Rendicion(persona = "Juan",
			fecha = hoy,
			total = 100),
		new Rendicion(persona = "Juan",
			fecha = new Date().minusMonths(1),
			total = 250),
		new Rendicion(persona = "Juan",
			fecha = new Date().minusDays(1),
			total = 120),
		new Rendicion(persona = "Ricky",
			fecha = hoy,
			total = 500),
		new Rendicion(persona = "Ricky",
			fecha = new Date().minusDays(2),
			total = 50)
	]

	}
```

Para sacar los totales por cliente, el Dictionary es una estructura auxiliar que nos sirve particularmente, porque permite acceder en forma directa al cliente que estoy buscando. El único tema es que la primera vez que averigüe el total de un cliente no va a estar en esa estructura, entonces debo asumir que el total es 0:

```
method totalPorCliente() {
	const totales = new Dictionary()
	rendiciones.forEach{rendicion =>
		var total = totales.getOrElse(rendicion.persona(), { 0 })
		total += rendicion.total()
		totales.put(rendicion.persona(), total)
	}
	return totales
}
```

Entonces de la lista generamos un conjunto de pares clave - valor, donde tenemos

- Persona
- Total

Lo mismo podemos hacer para la fecha:

```
method totalPorFecha() {
	const totales = new Dictionary()
	rendiciones.forEach{ rendicion =>
```

```
            var total = totales.getOrElse(rendicion.fecha(), { 0 })
            total += rendicion.total()
            totales.put(rendicion.fecha(), total)
        }
        return totales
}
```

Tenemos este método que nos ayuda a mostrar los resultados por pantalla, no es algo que queramos que hagan pero para este ejemplo el test es un poco engorroso de armar:

```
method mostrarTotalesPorCliente() {
        self.totalPorCliente().forEach({ persona, total =>
            console.println(persona + " - $ " + total)
        })
}
```

El forEach de un Dictionary recorre cada par de elementos clave y valor.

En general, los Dictionaries (también llamados mapas en otros lenguajes) son estructuras útiles para generar totales o bien para hacer cachés de valores, por su manera cómoda de acceder por clave.

6 Resumen

A lo largo de este capítulo hemos repasado el ejemplo de manejo de colecciones, ahora presentando las clases Set, List y Dictionary. La interfaz de las colecciones no han cambiado, seguimos teniendo mensajes poderosos como map, filter, sum, find, etc. que nos permiten subir la declaratividad de nuestras soluciones, valiéndonos para ello de los bloques de código (o lambdas), que nos permiten definir comportamiento anónimo.