Machete de sintaxis Wollok[1]

Elementos Comunes

Sintaxis básica

Objeto con un atributo y dos métodos	`object pepita {`
	`  var energia = 0`
	
	`  method volar(kilometros) {`
	`  energia += kilometros`
	`  }`
	
	`  method puedeVolar() {`
	`  return energia > 0`
	`  }`
	`}`

Comentario	`// un comentario`
	`/* un comentario`
	`  multilínea */`

Strings	`"uNa CadEna"` `'uNa CadEna'`
Booleanos	`true false`
Conjunto	`#{}` `#{1, 2}`
Lista	`[]` `[1, 1, 2]`
Bloques sin parámetros	`{algo}`
Bloques / Exp. lambda (De un parámetro)	`{x => algo con x}`
Bloques / Exp. lambda (Más de un parámetro)	`{x, y => algo con x e y}`

Operadores lógicos y matemáticos

Equivalencia	`==`
Identidad	`===`
~ Equivalencia	`!=`
Comparación de orden	`> >= < <=`
Disyunción (O lógico)	`||` `or`

[1] Este apunte está basado en la *Guía de lenguajes 3.1.1*, elaborada por docentes de la cátedra Paradigmas de Programación de la Universidad Tecnológica Nacional, Facultad Regional Buenos Aires.

Conjunción (Y lógico)	`&&` `and`

Negación	`! unBool`
	`unBool.negate()`
	`not unBool`

Operadores aritméticos	`+ - * /`
División entera	`dividendo.div(divisor)`
Resto	`dividendo % divisor`
Valor absoluto	`unNro.abs()`
Exponenciación	`base ** exponente`
Raíz cuadrada	`unNro.squareRoot()`
Máximo entre dos números	`unNro.max(otroNro)`
Mínimo entre dos números	`unNro.min(otroNro)`
Verificar si un número está entre otros dos	`unNro.between(uno, otro)`
Par	`unNro.even()`
Impar	`unNro.odd()`

Operaciones simples sin efecto sobre colecciones / listas

Longitud	`coleccion.size()`
Si está vacía	`coleccion.isEmpty()`
Concatenación	`coleccion + otraColeccion`
Unión	`set.union(coleccion)`
Intersección	`set.intersection(coleccion)`
Acceso por índice	`lista.get(indice)` *(base 0)*
Pertenencia	`coleccion.contains(elem)`
Máximo	`coleccionOrdenable.max()`
Minimo	`coleccionOrdenable.min()`
Sumatoria	`coleccionNumerica.sum()`
Aplanar	`coleccionDeColecciones.flatten()`

Primeros n elementos	`lista.take(n)`
Primer elemento	`lista.head()` `lista.first()`
Último elemento	`lista.last()`
Sin repetidos	`coleccion.asSet()`

Operaciones avanzadas (de orden superior) sin efecto sobre colecciones/listas

Sumatoria según transformación	`coleccion.sum(bloqueNumericoDe1)`
Filtrar	`coleccion.filter(bloqueBoolDe1)`
Transformar	`coleccion.map(bloqueDe1)`
Todos cumplen (true para lista vacía)	`coleccion.all(bloqueBoolDe1)`
Alguno cumple (false para lista vacía)	`coleccion.any(bloqueBoolDe1)`
Transformar y aplanar	`coleccion.flatMap(bloqueDe1)`
Reducir/plegar a izquierda	`coleccion.fold(valorInicial, bloqueDe2)`
Reducir/plegar a derecha	`NA`
Apareo con transformación	`NA`
Primer elemento que cumple condición	`coleccion.find(bloqueBoolDe1)` `coleccion.findOrElse( bloqueBoolDe1, bloqueSinParametros)`
Cantidad de elementos que cumplen condición	`coleccion.count(bloqueBoolDe1)`
Obtener colección ordenada.	`coleccion.sortedBy(bloqueBoolDe2)`
Máximo según criterio.	`coleccion.max(bloqueOrdenableDe1)`
Mínimo según criterio.	`coleccion.min(bloqueOrdenableDe1)`

Mensajes de colecciones con efecto

Agregar un elemento.	`coleccion.add(objeto)`
Agregar todos los elementos de la otra colección	`coleccion.addAll(otraColeccion)`
Evaluar el bloque para cada elemento.	`coleccion.forEach(bloqueConEfectoDe1)`

Eliminar un objeto.	`coleccion.remove(objeto)`
Eliminar todos los elementos.	`coleccion.clear()`
Deja ordenada la lista según un criterio.	`lista.sortBy(bloqueBoolDe2)`

Guía básica de Colecciones y Closures
Programación con Objetos 1
Versión del 10/08/2018

Las colecciones son objetos que agrupan a otros. Su comportamiento es el esperado para un conjunto o lista de objetos. No conoces nada acerca de los objetos de dominio, por eso, para que puedan interactuar se necesitan de los objetos Closures, también conocidos como bloques o lambdas.

1. Closures

Un closure[1] permite convertir una porción de código en un objeto. El beneficio de tener una porción de código como objeto es que se puede diferir su ejecución: El lugar donde se escribe el código puede no ser el mismo donde se pide la ejecución. Incluso, el código podría no ejecutarse nunca.

1. Para construir un closure básico, se usa `{=>codigo}` Por ejemplo, `{=>pepita.comer(10)}`. Escribir ese código en el REPL y revisar que sucede. ¿Es un objeto? ¿Pepita comió luego de esa línea?[2].

2. Si se quiere ejecutar el código interno, se debe utilizar el mensaje `apply()`.
   Ejecutar en el REPL cada una de estas líneas y analizar las diferencias:
   ```
   {=>pepita.energia()}
   pepita.energia()
   {=>pepita.energia()}.apply()
   ```
   El método `apply()` ejecuta el código y devuelve su resultado. En el ejemplo anterior, lo último que se ejecutó fue la energía que tenía pepita en ese momento.

3. La siguiente ejecución devuelve 4. Coloquialmente se dice que el *bloque devuelve 4* aunque lo estrictamente correcto sería decir que *el método `apply()` del closure devuelve 4*.
   ```
   {=>10 + 10
   pepita.energia()
   "hola".length()}.apply()
   ```
   Como la versión actual del REPL solo acepta código de una línea, ese código no se puede ejecutar directamente desde ahí. Para probarlo se puede escribir un objeto `ejemplo` con un método `bloqueDePrueba()` que devuelva el closure.
   ```
   object ejemplo {
   ```

[1] En otras tecnologías el concepto es conocido como Lambda o Bloque de código
[2] Se puede verificar ejecutando `pepita.energia()`

```
    method bloqueDePrueba() {
        return {=>10 + 10
            pepita.energia()
            "hola".length()}
    }
}
```

Y desde el REPL:

```
ejemplo.bloqueDePrueba().apply()
```

4. En el caso de que la última línea de código del bloque haya sido una orden (no devuelve nada), entonces el bloque no devuelve nada.
   Probar en el REPL:
   `{=>pepita.comer(10)}`[3].

   ¿Hay polimorfismo entre un bloque que ejecuta una orden y un bloque que ejecuta una consulta?. La clave para responder esta pregunta está en pensar si el que envía el mensaje `apply()` puede tratarlos indistintamente.

5. Un bloque además puede recibir parámetros. Los parámetros se indican a la izquierda de la flecha `=>` y se separan por coma. No hay cantidad fija de parámetros que pueda recibir el bloque. Al enviar el mensaje `apply()` se debe indicar los valores de los parámetros.
   Ejecutar a través de un objeto de ejemplo:
   ```
   {unosGramos,unosKilometros => pepita.comer(unosGramos)
   pepita.volar(unosKilometros)
   pepita.energia()}.apply(100,5)
   ```
   ¿Hay polimorfismo entre un bloque que recibe un argumento y otro que no recibe ninguno?

2. Colecciones

Una colección es un objeto que agrupa otros objetos. Hay dos sabores de colecciones. Hay colecciones que son *conjuntos* y otras que son *listas*. Un conjunto es una agrupación que no tiene un orden. Mientras que en una lista sí. Además, en una lista puedo incluir a un objeto más de una vez. Por ejemplo, que el mismo objeto ocupe la posición 3 y la 7. En un conjunto no tiene sentido que un objeto figure más de una vez, alcanza con saber si pertenece o no.

1. Un conjunto se construye encerrando las referencias de los objetos separadas por coma entre {# y }. Por ejemplo, el conjunto de los número primos menores que 20: `#{2, 3, 5, 7, 11, 13, 17, 19}`. Ingresar ese conjunto en el REPL. El orden en que se muestra es el que a Wollok le queda más cómodo y nunca se debe asumir que es conocido por el programador.

[3] quizás deba probar `{=>pepita.comer(puniadoDeAlpiste)}` en el caso que utilice la versión de pepita que come distintos alimentos

2. Una lista se construye encerrando las las referencias de los objetos separados por coma entre corchetes. Por ejemplo, la lista de las comidas del almuerzo de la semana laboral, ordenados de lunes a viernes `["canelones", "milanesa", "ravioles", "milanesa","milanesa"]`. Probarlo en el REPL.

3. ¿Qué sucede si en el ejemplo anterior se utiliza un conjunto?
   `#{"canelones", "milanesa", "ravioles", "milanesa","milanesa"}`

2.1. Mensajes básicos que entienden las colecciones.

2.1.1. Mensajes sin efecto (consultas).

1. Para saber el tamaño: `coleccion.size()`.
   Probar:
   ```
   #{5,4,2}.size()
   [5,4,2].size()
   [1,2,3,3,3,3].size()
   #{1,2,3,3,3,3}.size()
   ```

2. Para saber si está vacía: `coleccion.isEmpty()`.
   Probar:
   ```
   #{5,4,2}.isEmpy()
   [].isEmpty()
   ```

3. Para saber si un elemento está incluido o no: `coleccion.contains(elemento)`.
   Probar:
   ```
   #{5,4,2}.contains(4)
   [5,4,2].contains(2)
   #{5,4,2}.contains(1)
   ```

4. En ciertas ocaciones se necesita convertir una lista en un conjunto o viceversa. También, puede ocurrir que se tenga una colección sin saber si se trata de una lista o un conjunto y se necesite trabajar a ese nivel de abstracción. Por eso todas las colecciones entienden los mensajes `asList()` y `asSet()`. En el caso de enviar `asList()` a una lista o `asSet()` a un conjunto, se devuelve la misma colección. En caso contrario, devuelve una nueva colección del otro tipo con los mismos elementos. Ejemplos a probar:
   ```
   #{5,4,2}.asList()
   [5,4,2].asSet()
   [5,2,4,2].asSet()
   [5,2,4,2].asList()
   #{5,4,2}.asSet()
   ```

2.1.2. Mensajes con efecto (órdenes)

Para probar las órdenes se necesita generar un objeto de ejemplo que tenga un atributo que apunte a una colección. Luego de enviar el mensaje, se verifica el efecto realizando una consulta.

1. Agregar un elemento: `coleccion.add(element)`.

```
object ejemplo{
     var col = [1,2,3]
     method col() {
          return col
     }
     method agregar(elemento) {
          col.add(elemento)
     }
}
```

   Probar en el REPL:

```
ejemplo.agregar(4)
ejemplo.col()
ejemplo.agregar(5)
ejemplo.col()
ejemplo.agregar([1,2,3])
ejemplo.col()
```

2. Probar el ejemplo anterior usando un conjunto dentro del objeto ejemplo en lugar de una lista.

3. Agregar más de un elemento a la vez: `coleccion.addAll(otraCollecion)`
   Modificar el objeto ejemplo de la siguiente manera:

```
object ejemplo{
     var col = [1,2,3]
     method col() {
          return col
     }
     method agregarVarios(elementos) {
          col.addAll(elementos)
     }
}
```

   Probar en el REPL las siguientes líneas:

```
ejemplo.agregarVarios([2,3,4,5,6])
ejemplo.col()
ejemplo.agregarVarios(#{2,3,4,5,6})
ejemplo.col()
ejemplo.agregarVarios(4)
```

```
ejemplo.col()
```

4. Probar el ejemplo anterior usando un conjunto dentro del objeto ejemplo en lugar de una lista.

5. Así como existen los métodos `add(e)` y `addAll(c)`, se pueden eliminar objetos usando `coleccion.remove(elemento)` y `coleccion.removeAll(otraColeccion)`

```
object ejemplo{
     var col = [1,2,3]
     method col() {
          return col
     }
     method eliminar(elemento) {
          col.remove(elemento)
     }
     method eliminarVarios(elementos) {
          col.removeAll(elementos)
     }
}
```

Probar en el REPL:

```
ejemplo.eliminarVarios(1)
ejemplo.eliminar(9)
ejemplo.col()
ejemplo.eliminar(1)
ejemplo.col()
ejemplo.eliminarVarios([3,2])
ejemplo.col()
```

6. Probar el ejemplo anterior usando un conjunto dentro del objeto ejemplo en lugar de una lista.

3. Colecciones y bloques

Uno de los principales usos de las colecciones es enviar el mismo mensaje a varios objetos y combinar sus respuestas según sea la necesidad. El mensaje a enviar se indica a través de un *closure* que se pasa por parámetro a la colección. A su vez, el elemento sobre el cual se envía el mensaje se le indica al closure también por parámetro.

3.1. Comandos

Una colección puede contener un conjunto de objetos que entienden mensajes que son órdenes. Una orden invoca un método que tiene efecto, es decir, modifica el estado del objeto. Para enviar la misma orden a todos los elementos de una colección se necesita un comando. En este contexto, un comando es un *closure* que recibe por parámetro el objeto al cual se le quiere enviar la orden. Cuando el closure es ejecutado se envía el mensaje al objeto en cuestión. Un comando no devuelve nada.

La manera de ejecutar el comando sobre cada uno de los elementos de la colección es: `coleccion.forEach(comando)`.
Ejecutar el siguiente código en el REPL:

```
#{pepita,pepon,pipa}.forEach({unAve => unAve.comer(10)})
pepita.energia()
pepon.energia()
pipa.caloriasIngeridas()
```

3.2. Condiciones

Una condición es una afirmación sobre un objeto. Ésta tiene un valor de verdad: puede ser verdadero o falso. Para escribir una condición en wollok se usa un bloque que recibe por parámetro un objeto y devuelve un booleano. Ej.: `{unNumero => unNumero < 10 }` es una condición para saber si un número es menor que diez. Hay varios problemas que se pueden resolver combinando colecciones y condiciones.

1. Para saber si un objeto de la colección cumple una condición: `coleccion.any(condicion)`.
   Ejecutar las siguiente líneas para saber si hay algún número menor a 10 en la colección:
   ```
   [1,4,10,21].any({unNumero => unNumero < 10 })
   #{1,10,21}.any({unNumero => unNumero < 10 })
   #{10,21}.any({unNumero => unNumero < 10 })
   #{pepita,pepon}.any({unNumero => unNumero < 10 })
   [ ].any({unNumero => unNumero < 10 })
   ```

2. Para saber si todos los objetos de la colección cumplen una condición:
   `colección.all(condicion)`.
   Ejecutar las siguiente líneas para saber si todos los números son menores que 10 en la colección:
   ```
   [1,4,10,21].all({unNumero => unNumero < 10 })
   [1,4,5].all({unNumero => unNumero < 10 })
   #{1,10,21}.all({unNumero => unNumero < 10 })
   #{pepita,pepon}.all({unNumero => unNumero < 10 })
   #{ }.all({unNumero => unNumero < 10 })
   ```

3. Para obtener un objeto de la colección que cumpla una condición:
   `coleccion.find(condicion)`.
   Ejecutar las siguiente líneas para encontrar un número menor a 10 en la colección:
   ```
   [1,4,10,21].find({unNumero => unNumero < 10 })
   #{1,10,21}.find({unNumero => unNumero < 10 })
   #{15,21}.find({unNumero => unNumero < 10 })
   #{pepita,pepon}.find({unNumero => unNumero < 10 })
   #{ }.find({unNumero => unNumero < 10 })
   ```

4. Para obtener un subconjunto/sublista de la colección que cumpla una condición:
   `coleccion.filter(condicion)`.
   Ejecutar las siguientes líneas para encontrar todos los número menores a 10 en la colección:
   ```
   [12,1,4,10,21].filter({unNumero => unNumero < 10 })
   #{1,10,21}.filter({unNumero => unNumero < 10 })
   #{15,21}.filter({unNumero => unNumero < 10 })
   #{pepita,pepon}.filter({unNumero => unNumero < 10 })
   #{ }.filter({unNumero => unNumero < 10 })
   ```

3.3. *Transformers*

Un *Transformer* es similar a una condición, pero se utiliza para obtener un objeto a partir de otro. Una condición puede ser vista como un caso particular de un *transformer* que convierte un objeto en un booleano. Por lo tanto, la manera de construir un *Transformer* en wollok es también con un bloque, que recibe un elemento por parámetro y devuelve otro. El mensaje más importante de colecciones que usa un transformer es *map(transformer)*, utilizado para convertir una colección de elementos en otra cuyos items son elementos derivados de los primeros.

Se utiliza así: `colección.map(transformer)`.
Esta línea convierte los números en sus dobles:
```
[1,4,10,21].map({unNumero => unNumero * 2 })
```
Esta línea permite saber los entrenadores de las aves:
```
#{pepita,pepon,pipa}.map({unAve => unAve.entrenador()})
```

Probar los ejemplos en el REPL.

4. Otros mensajes útiles de las colecciones

Hay otros mensajes útiles en colecciones. La mayoría de los ejercicios de la materia van a hacer uso de los desarrollados en las secciones anteriores. Sin embargo, en ocaciones pueden resultar útiles.

4.1. Con efecto

1. `unaColeccion.clear()` Remueve todos los objetos de la colección
   Probar a través de un objeto de ejemplo:

```
method prueba() {
     var c = [1,2,3]
     c.clear()
     return c
}
```

4.2. Sin efecto

1. `unaColeccion.sum()` Devuelve la suma de los objetos de la colección (los objetos deben entender el mensaje +).
   Probar:
   `#{1,2,3}.sum()`

2. `unaColeccion.sum(transformer)` Aplica una transformación a cada objeto y luego los suma.
   Probar en el REPL este código para saber la cantidad de caracteres totales de la colección:
   `#{"Hola","Mundo","Wollok"}.sum({x=>x.length()})`

3. `unaColeccion.min()` Devuelve el mínimo de los objetos de la colección (los objetos deben ser comparables).
   Probar:
   `#{5,1,2,3}.min()`

4. `unaColeccion.min(transformer)` Aplica una transformación a cada objeto y luego calcula el mínimo.
   Probar en el REPL este código para saber la palabra más corta de la colección:
   `#{"Hola","Mundo","Wollok"}.min({x=>x.length()})`

5. `unaColeccion.max()` Devuelve el máximo de los objetos de la colección (los objetos deben ser comparables).
   Probar:
   `#{1,2,3,0}.max()`

6. `unaColeccion.max(transformer)` Aplica una transformación a cada objeto y luego el máximo.

Probar en el REPL este código para saber la palabra más corta de la colección:
`#{"Hola","Mundo","Wollok"}.max({x=>x.length()})`

7. `unaColeccion.anyOne()`. Devuelve un objeto cualquiera contenido en la colección.
   Probar:
   `#{"Hola","Mundo","Wollok"}.anyOne()`

8. `unaColeccion.count(condicion)` Devuelve la cantidad de objetos que cumplen la condición.
   Probar:
   `#{1,5,12,3,32}.count(unNumero => unNumero < 10)`

9. La ejecución del método `find(condicion)` lanza error si el objeto no se encuentra. En el caso que se sepa cual es el valor por defecto se puede usar el mensaje `coleccion.findOrDefault(condicion,default)`.
   Ejecutar la siguiente linea para encontrar un numero menor a 10 en la colección o 9 si no había
   ```
   [1,4,10,21].findOrDefault({unNumero => unNumero < 10 },9)
   #{1,10,21}.findOrDefault({unNumero => unNumero < 10 },9)
   #{15,21}.findOrDefault({unNumero => unNumero < 10 })
   #{pepita,pepon}.findOrDefault({unNumero => unNumero < 10 },9)
   #{ }.findOrDefault({unNumero => unNumero < 10 },9)
   ```

10. `unaColeccion.findOrElse(condicion,bloqueDefault)`: Similar a `findOrDefault(condicion)`, con la diferencia que el valor por defecto es el resultado de ejecutar el bloque recibido por parámetro. Esto es útil para casos en el cual el valor por defecto puede ser un cálculo complejo y solo se quiere ejecutar en caso de ser necesario.
    Probar:
    ```
    #{1,10,21}.findOrElse({unNumero => unNumero < 10 },{=>pepe.sueldo()})
    #{15,21}.findOrElse({unNumero => unNumero < 10 },{=>pepe.sueldo()})
    ```

11. `unaColeccion.occurrencesOf(elemento)`: Devuelve la cantidad de veces que está ese elemento en la colección. Es un mensaje que tiene más sentido en las listas, pero los conjuntos también saben contestarlo.
    Probar:
    `[1,2,10,21,2,2,2].occurrencesOf(2)`

12. Concatenación: Con el operador + se devuelve una nueva colección con todos los elementos de ambas colecciones:
    Probar:
    ```
    #{1,2} + #{3,2}
    [1,2] + [3,2]
    #{1,2} + [3,2]
    [1,2] + #{3,2}
    ```

13. Igualdad: Las colecciones entienden el operador ==. Son iguales si son el mismo tipo de colección y contienen los mismos elementos. En el caso de la lista además se fija el orden.
    Probar:
    ```
    #{1,2} == #{1,2}
    #{2,1} == #{1,2}
    #{1,2} == #{3,2}
    [1,2] == [1,2]
    [1,2] == [2,3]
    [1,2] == [2,1]
    #{1,2} == [1,2]
    [1,2] == #{1,2}
    ```

14. En caso de tener una colección de colecciones, se puede obtener una única colección que contenga solo los elementos de las colecciones internas. Para eso se utiliza el mensaje `coleccion.flatten()`.
    Probar:
    ```
    [[1, 2, 3], [3, 4]].flatten()
    [[1, 2, 3], #{3, 4}].flatten()
    [#{1, 2, 3}, #{3, 4}].flatten()
    #{[1, 2, 3], [3, 4]}.flatten()
    ```

4.2.1. Fold

Hay casos que debido a su complejidad, ninguno de los mensajes sin efecto anteriores alcanza para resolver el problema. Esos casos complejos se pueden resolver utilizando el mensaje `fold(objetoInicial, unClosure)`. A través de este mensaje se puede realizar un cálculo haciendo que todos los objetos de la colección colaboren.

El algoritmo funciona de la siguiente manera: Se realiza un cálculo (representado por el closure) entre el primer elemento de la colección, y el objeto inicial pasado por parámetro. El resultado de ese cálculo se toma para repetir el cálculo entre dicho valor y el segundo elemento de la colección. De esa manera se van enganchando el resultado del cálculo de un elemento con el elemento siguiente hasta completar el cálculo con todos los elementos.

Es por eso que el bloque recibido por parámetro, que representa el cálculo a realizar, recibe dos parámetros: el primer parámetro es el valor inicial o el acumulado (según se trate del primer elemento o de uno subsiguiente). El segundo parámetro es el elemento de la colección.

A través de los siguentes ejemplos se puede comprender como es el funcionamiento. Probarlos en el REPL.

a) La siguiente línea permite calcular la sumatoria de una colección de números:
   `[1,5,7,9].fold(0,{acum,item=>acum+item})`
   Análisis paso por paso de como se resuelve el problema:

- Primero el bloque es ejecutado con los parámetros 0 (valor inicial) y 1 (primer elemento de la lista). El bloque los suma y lo devuelve. El resultado es 1.
- Luego el bloque es ejecutado con los parámetros 1 (por ser el valor acumulado hasta el momento) y 5 (por ser el segundo elemento de la lista). El bloque los suma y los devuelve. El resultado es 6.
- Luego el bloque es ejecutado con los parámetros 6 (por ser el valor acumulado hasta el momento) y 7 (por ser el tercer elemento de la lista). El bloque los suma y los devuelve. El resultado es 13.
- Finalmente el bloque es ejecutado con los parámetros 13 (por ser el valor acumulado hasta el momento) y 9 (por ser el cuarto elemento de la lista). El bloque los suma y los devuelve. El resultado es 22.
- Como no hay más elementos, el resultado final es 22.

b) No es necesario que ambos parámetros del bloque sean objetos del mismo tipo. El siguiente ejemplo calcula la cantidad de caractares totales que hay en un string:
`["Hola","Mundo","Wollok"].fold(0,{acum,item=>acum+item.length()})`
El parámetro `acum` del bloque es un número, el parámetro `item` es un string.

c) Otro ejemplo un poco más complejo: Calcular la palabra más larga de una lista de string.

```
var col = ['Hola','Mundo','Wollok','Programación','Objetos']
col.fold(col.anyOne(),
          {maxActual,item =>
          if (maxActual.length() > item.length()) maxActual else item})
```

El objeto inicial puede ser cualquiera de la colección. El *if* es usado como una expresión para devolver el mayor entre dos elementos.

5. Mensajes que sólo entienden las listas

Las listan mantienen los elementos ordenados, por eso hay una serie de mensajes asociados a esa característica que no son válidos para los conjuntos. El primer elemento de una lista está en la posición 0.

5.1. Con efecto

1. Para reordenar los elementos de la lista se puede usar un comparador que determina si un elemento es menor que otro. En wollok, un comparador es un *closure* que recibe dos elementos y devuelve verdadero si el primero es menor.
   La manera de usarse es `lista.sortBy(comparador)`.
   El siguiente ejemplo ordena las palabras por su cantidad de letras:
   `["Independiente", "Club", "Atlético"].sortBy({a,b => a.length() < b.length()})`

5.2. Sin efecto

1. Para acceder a un elemento que se encuentra en una posición se usa `lista.get(posicion)`.
   Probar:
   ```
   ['a','b','c'].get(0)
   ['a','b','c'].get(1)
   ['a','b','c'].get(2)
   ['a','b','c'].get(3)
   ['a','b','c'].get(-1)
   ```

2. Para acceder al primer o último elemento se puede usar `lista.first()` y `lista.last()`.
   Probar:
   ```
   ['a','b','c'].first()
   ['a','b','c'].last()
   [ ].first()
   [ ].last()
   ```

3. Para obtener una sublista con todos los elementos entre dos posiciones `lista.subList(posicionInicial, posicionFinal)`.
   Probar:
   ```
   ['a','b','c'].sublist(1,2)
   ['a','b','c'].sublist(0,2)
   ['a','b','c'].sublist(-100,75)
   ['a','b','c'].sublist(1,1)
   ```

4. Para obtener los primeros n elementos de una lista se usa `lista.take(n)`.
   Probar:
   ```
   ['a','b','c'].take(1)
   ['a','b','c'].take(2)
   ['a','b','c'].take(0)
   ['a','b','c'].take(100)
   ['a','b','c'].take(-9)
   ```

5. Para obtener todos los elementos descartando los primeros n elementos se usa `lista.drop(n)`.
   Probar:
   ```
   ['a','b','c'].drop(1)
   ['a','b','c'].drop(2)
   ['a','b','c'].drop(0)
   ['a','b','c'].drop(100)
   ['a','b','c'].drop(-9)
   ```

6. Para obtener otra lista con los mismos elementos pero en el orden inverso se usa `lista.reverse()`.
   Probar:
   ```
   ['a','b','c'].reverse() ['a'].reverse()
   [ ].reverse()
   ```

6. Mensajes que sólo entienden los conjuntos

6.1. Sin efecto

1. `conjunto.union(otroConjunto)` Devuelve la unión de dos conjuntos. Es similar al operador +.
   ```
   #{1,2,3,4}.union(#{2,5})
   #{1,2,3,4}.union(#{ })
   #{1,2,3,4}.union(2)
   ```

2. `conjunto.intersection(otroConjunto)` Devuelve la intersección de dos conjuntos (los elementos en común).
   ```
   #{1,2,3,4}.intersection(#{2,5})
   #{1,2,3,4}.intersection(#{2,3,5 })
   #{1,2}.intersection(#{3,4 })
   #{1,2}.intersection(2)
   ```

3. `conjunto.difference(otroConjunto)` Devuelve un conjunto con los elementos del primero que no están en el segundo conjunto.
   ```
   #{1,2,3,4}.difference(#{2,3})
   #{1,2,3,4}.difference(#{2,3,5 })
   #{1,2}.difference(#{3,4 })
   #{1,2}.difference(2)
   ```

A. Machete Oficial de Closures y Colecciones

Closures				
Tipo	**Parámetros**	**Retorno**	**Detalle**	**Ejemplo**
Comando	1	Nada	Ejecuta una orden sobre el parámetro	`{ave=>ave.comer(10)}`
Condición	1	true/false	Usado para saber si el parámetro cumple o no una condición	`{ave=>ave.energia()>0}`
Transformer	1	otro objeto	Usado para obtener un objeto a partir del parámetro	`{ave=>ave.entrenador()}`
Comparador	2	true/false	Es un criterio para ordenar elementos. Indica si el primer elemento debe ir antes que el segundo	`{a,b=> a.foo() < b.foo()}`

Tabla 1: Tipos de Closures

Constructores	
Colección	**Detalle**
Conjunto	`#{a,b,c,...,n}`
Lista	`[a,b,c,...,n]`

Tabla 2: Constructores de colecciones

Consultas		
Método	**Retorno**	**Detalle**
`c.contains(unObjeto)`	true/false	Si `unObjeto` está en `c`
`c.isEmpty()`	true/false	Si `c` no tiene elementos
`c.size()`	un número	La cantidad de elemento de `c`
`c.asList()`	una lista	una lista con todos los elementos de c (puede ser el mismo objeto c si ya era una lista)
`c.asSet()`	un conjunto	un conjunto con todos los elementos de c (puede ser el mismo objeto c si ya era un conjunto)
`c.any(unaCondicion)`	true/false	Si hay algún objeto en `c` que cumple con `unaCondicion`

Método	Retorno	Detalle
`c.all(unaCondicion)`	true/false	Si todos los objetos en `c` cumplen con `unaCondicion`
`c.find(unaCondicion)`	un objeto	Devuelve un objeto de `c` que cumple con `unaCondicion`. Si no encuentra lanza error
`c.findOrDefault(unaCondicion,unObjeto)`	un objeto	Devuelve un objeto de `c` que cumple con `unaCondicion`. Si no hay ninguno devuelve `unObjeto`
`c.findOrElse(unaCondicion,unClosure)`	un objeto	Devuelve un objeto de `c` que cumple con `unaCondicion`. Si no hay ninguno ejecuta `unClosure` y devuelve lo que dicho bloque retorne. `unClosure` no tiene parámetros
`c.filter(unaCondicion)`	otra colección	Devuelve todos los objetos en `c` que cumplen con `unaCondicion`
`c.map(unTransformer)`	otra colección	Aplica `unTransformer` a todos los elementos de `c`. Devuelve todos los objetos que devolvió el transformer
`c.sum()`	un número	Devuelve la suma de todos los elementos de `c`
`c.sum(unTransformer)`	un número	Devuelve la suma de todos los elementos transformados de `c`
`c.min()`	un número	Devuelve el elemento mínimo de `c`
`c.min(unTransformer)`	un objeto	Devuelve el mínimo de los elementos transformados de `c`
`c.max()`	un número	Devuelve el elemento máximo de `c`
`c.max(unTransformer)`	un objeto	Devuelve el máximo de los elementos transformados de `c`
`c.anyOne()`	un objeto	Devuelve un objeto cualquiera de `c`
`c.count(unaCondicion)`	un número	Devuelve la cantidad de elementos de `c` que cumplen `unaCondicion`

Método	Retorno	Detalle
`c.ocurrenceOf(unObjeto)`	un número	Devuelve la cantidad de veces que está `unObjeto` en `c`
`c + otraColeccion`	una colección	Devuelve una colección que tiene todos los elementos de `c` y de `otraColeccion`
`c == otraColeccion`	true/false	Compara si `c` y `otraColeccion` son del mismo tipo y tienen los mismos elementos. Si es una lista también se fija que estén en el mismo orden
`colDeColecciones.flatten()`	una colección	`c` debe ser una colección de colecciones. Devuelve una colección con los elementos de las colecciones internas
`c.fold(inicial,unClosure)`	un objeto	`unClosure` recibe dos parametros, el primero representa a un valor intermedio. El segundo representa a un objeto de la colección. La primera vez el bloque se ejecuta utilizando `inicial` como valor intermedio. El bloque devuelve el próximo valor intermedio a ser utilizado con el siguiente elemento de la lista. El último valor intermedio es el considerado final y es lo que retorna el método. Ejemplo: contar los caracteres de una lista de Strings. `c.fold(0,{acum,item => acum + item.length()})`
Consultas Solo Para Listas		
Método	**Retorno**	**Detalle**
`lst.get(poiscion)`	un objeto	El elemento de `lst` que está en `posicion`. El primer elemento está en la posición 0.
`lst.first()`	un objeto	El primer elemento de `lst`
`lst.last()`	un objeto	El último elemento de `lst`

Método	**Retorno**	**Detalle**
`lst.sublist(posInicial,posFinal)`	una lista	La sublista entre `posInicial` y `posFinal` de `lst`
`lst.take(n)`	una lista	Una sublista con los primeros `n` elementos de `lst`
`lst.drop(n)`	una lista	Una sublista sin los primeros `n` elementos de `lst`
`lst.reverse(n)`	una lista	Una lista con los mismos elementos de `lst` pero en el orden inverso
Consultas Sólo Para Conjuntos		
Método	**Retorno**	**Detalle**
`conj.union(otroConj)`	un conjunto	Un conjunto con todos los elementos de `conj` y de `otroConj`
`conj.intersection(otroConj)`	un conjunto	Un conjunto con todos los elementos de `conj` que también están en `otroConj`
`conj.difference(otroConj)`	un conjunto	Un conjunto con todos los elementos de `conj` que no están en `otroConj`

Tabla 4: Métodos de colecciones que no alteran a la misma

Órdenes	
Método	**Detalle**
`c.add(unObjeto)`	Agrega `unObjeto` a `c`
`c.addAll(unaColeccion)`	Agrega todos los elementos de `unaColeccion` a `c`
`c.remove(unObjeto)`	Elimina `unObjeto` de `c`
`c.removeAll(unaColeccion)`	Elimina todos los elementos de `unaColeccion` de `c`
`c.clear()`	Elimina todos los elementos de `c`
`c.forEach(unComando)`	Ejecuta `unComando` sobre todos los elementos de `c`. Si bien este método no tiene efecto, se considera una órden porque se usa para enviar mensajes con efecto.
Órdenes que sólo entienden las listas	
`lst.sortBy(comparador)`	Ordena los elementos de `lst` según el criterio aportado por `comparador`

Tabla 3: Métodos de colecciones que tienen efecto