

CURSO DE PROGRAMACION SCALA

Sesión 2

Sergio Couto Catoira
ingscc00@gmail.com

Índice

- > Más colecciones
 - Vector
 - Set
 - Listas 2
 - Tuplas
 - Map
- > Principios de la programación funcional 1 : Efectos de lado
- > Option
- > Either
- > Más testing unitario

Más colecciones

› Vector

- Óptima para acceso aleatorio
- Append => :+
- Prepend => +:
- Update => updated

› List

- Óptima para LIFO
- elemento +: lista || lista.+: (elemento)
- Lista :+ elemento || lista.:+ (elemento)

Y más colecciones

› Set

- No duplicados
- Añadir elementos => +
- Quitar => -
- Union (++), intersect, diff, contains etc..
- No ordenado => Ojo con pattern matching => .toSeq

› List

- Óptima para LIFO
- elemento +: lista || lista.+: (elemento)
- Lista :+ elemento || lista.:+ (elemento)

Ejercicios

- › Función que añada un elemento al final de una lista
 - `def addToList(list: List[Int], elem: Int): List[Int] = ???`
- › Función que compruebe si una lista de enteros es un palíndromo
 - `def isPalindrome(list: List[Int]) : Boolean = ???`

Y todavía más colecciones

› Tuplas

- Puede ser de entre 2 y 22 elementos (Tuple2, Tuple3 etc..)
- Se accede con underscore + numero (myTuple._2)
- Puede combinar tipos
- Válido para pattern matching

› Map

- Val romanos = Map(1 → "I", 2->"II", 3->"III", 4->"IV", 5->"V", 6->"VI", 7->"VII", 8->"VIII", 9->"IX", 10->"X")
- Contains, keys, values
- Recuperar valores:
 - › romanos(5)
 - › romanos.get(5)
- “Ejercicio”: ¿Cuál es la diferencia entre las dos formas de recuperar valores? ¿Qué pasa si recuperar un valor que no existe en el map?

Ejercicios

- > Imprime el mapa anterior o uno semejante de tal forma (no importa el orden)
- > ¿Eres capaz de hacerlo ordenado?

```
5 => V
10 => X
1 => I
6 => VI
9 => IX
2 => II
7 => VII
3 => III
8 => VIII
4 => IV
```

```
1 => I
2 => II
3 => III
4 => IV
5 => V
6 => VI
7 => VII
8 => VIII
9 => IX
10 => X
```

Pista: Cuando hablamos de sets, vimos como convertirlo a una colección ordenada. Dicha colección tiene el método `SortBy`, (tendrás que indicarle el elemento por el que quieras ordenar)

Efectos de lado

> Función pura vs impura

- `def sum (x:Int, y: Int) = x+y`
- `Def sum (x: Int, y:Int)= {
 val z = x+y
 println (s"$x + $y equals $z")
 z
}`

> Ejemplos de efectos de lado:

- Leer o escribir un fichero
- Invocar un servicio web
- Arrancar otro thread
- Lanzar una excepción
- Enviar un email
- Lanzar un misil

Consecuencias de los efectos de lado

- > Dificultad para mantener un programa
 - Comprender el programa
 - Pruebas del programa
 - Descubrir y solucionar bugs
 - Optimizar
 - Reutilizar

Como “evitarlos”

- > Imposible evitarlos, son necesarios
- > Objetivo: Descoplar la parte pura de la parte impura
- > Parte pura se encarga de definir qué efectos necesitamos ejecutar.
- > Parte impura los ejecuta
- > Programación monádica



Option

- › No aporta información del error
 - Compuesto por Some y None
 - Se indica el tipo que puede contener: Option[T]
 - Útil para pattern matching
 - Permite tratar casos especiales sin levantar un efecto de lado

Ejercicios sobre option

- › Ejercicio: Define una función que multiplique un número por un valor opcional. Si no se recibe valor opcional debe multiplicar por defecto 1.5
 - `def aplicaInteres(cant: Double, tipo: Option[Double]): Double = ???`
- › Ejercicio: Define la misma función con la cantidad también opcional. Ahora no siempre podrás dar un resultado, por lo que la salida es otro option
 - `def aplicaInteres(cant: Option[Double], tipo: Option[Double]): Option[Double] = ???`

Either

- › Similar a Option, pero aporta información del error
 - Compuesto por Some y None
 - Se indica los tipos que puede contener: `Either[U, V]`
 - Útil para pattern matching
 - Permite tratar casos especiales sin levantar un efecto de lado y aportando información del error
 - Por convención, el valor izquierdo[U] se utiliza para representar información del fallo

Ejercicios con Either

- > Redefine la función anterior de modo que aporte información del error
 - `def aplicaInteres(cant: Option[Double], tipo: Option[Double]): Either[String, Double] = ???`
- > Redefínela nuevamente suponiendo que ambos valores de entrada son recibidos con su propio mensaje de error. Si es necesario propaga esos mensajes
 - `def aplicaInteres (cant: Either[String, Double], tipo: Either[String, Double]: Either [String, Double] = ???`

Más test unitario

› Scalacheck

- Permite definir propiedades genéricas y generadores de pruebas

```
val genPositiveInteger = for (n <- Gen.choose(-500, 500)) yield n
```

```
"sum" should "work for all numbers" in {  
  forAll(genPositiveInteger, genPositiveInteger) { (n1: Int, n2: Int) =>  
    val result = sum(n1, n2)  
    println(s"${n1} + ${n2} equals: ${result}")  
    result shouldEqual n1 + n2  
  }  
}
```

Ejercicios

- › Función que devuelva el penúltimo elemento de una lista
 - `def penultimate(list: List[Int]): Option[Int] = ???`
- › Función que duplica cada elemento de la lista x veces
 - `def duplicate(list: List[Int], x: Int) : List[Int]`
- › Función que haga rotar una lista de enteros x lugares hacia la izquierda
 - `def rotate(list: List[Int], x: Int): List[Int]`
 - › `rotate(List(1,2,3,4,5), 2) = List(3,4,5,1,2)`
 - › `rotate(List(1,2,3,4,5), -2) = List(4,5,1,2,3)`
- › Función que compruebe si una palabra es un palíndromo
 - `def isPalindrome(word: String): Boolean = ???`