UNIVERSIDAD COMPLUTENSE DE MADRID

ntic master School

Scala

Programación Funcional

Noviembre de 2023

Agenda

- **Funciones de orden superior**
- **Características superiores**
- **Funciones de orden superior destacadas**

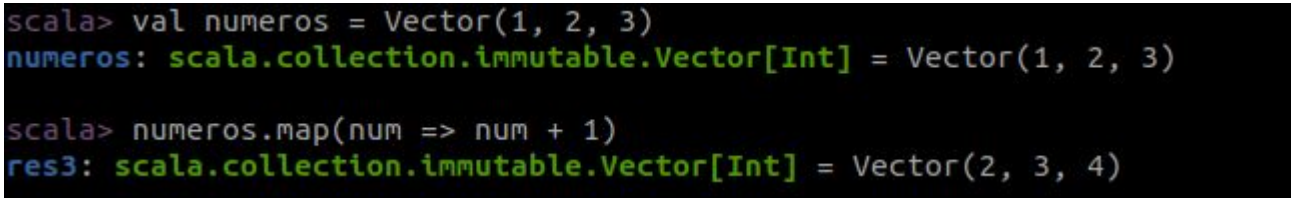
1 Funciones de orden superior

Colecciones funcionales

- La librería de colecciones de Scala contienen gran cantidad de funciones de orden superior:
  - Métodos que esperan una función o funciones como argumentos.
  - Métodos que devuelven una función.

- Qué se considera una función: Cualquier cosa que puede ser invocado:
  - Toma cero o más argumentos.
  - Una función devuelve un valor.

Orden superior == Mayor abstracción

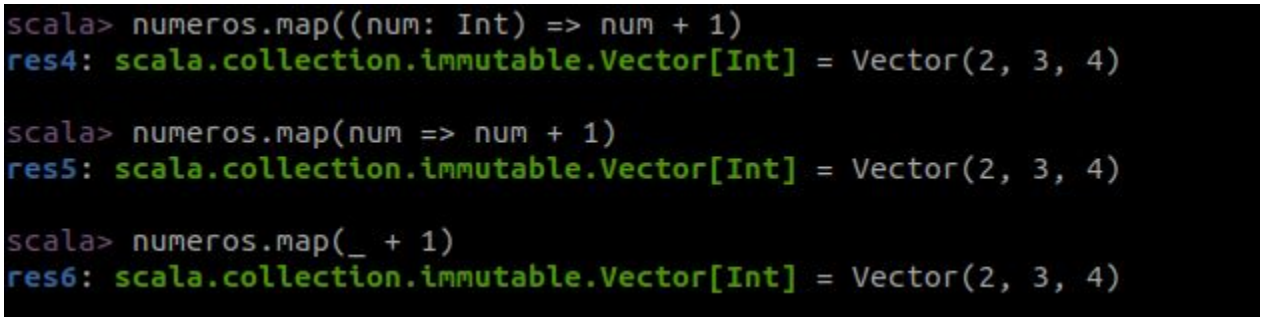


- Las funciones de orden superior permiten elevar el nivel de abstracción.
  - Se dice el ***qué*** hacer (programación declarativa)
  - No se especifica el ***cómo*** se debe hacer.

- Por ejemplo: función ***map***: espera una función como argumento y es aplicada a cada elemento de la colección y devuelve su valor.

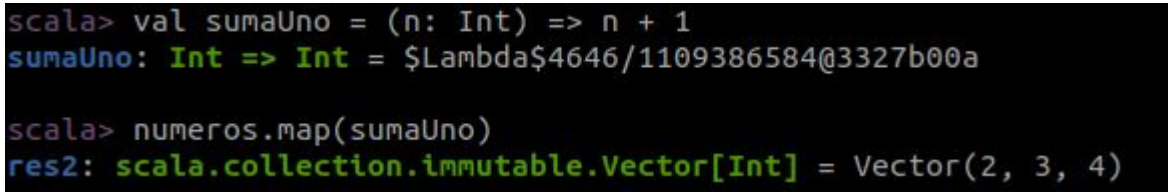
2 Características superiores

Funciones literales



- Es una función anónima:
  - Útil para pasar funciones como argumento a un método.
- Notación corta:
  - Cada guión bajo (_) indica el argumento de la función.
  - Sólo se puede usar si se usa cada parámetro una única vez

Valores función



- En Scala las funciones son de ***primera clase*** -> Una función se considera un objeto.
- Una función se puede asignar a una variable.
- Un valor función puede ser un argumento de una función de orden superior.

Los tipos de una Función

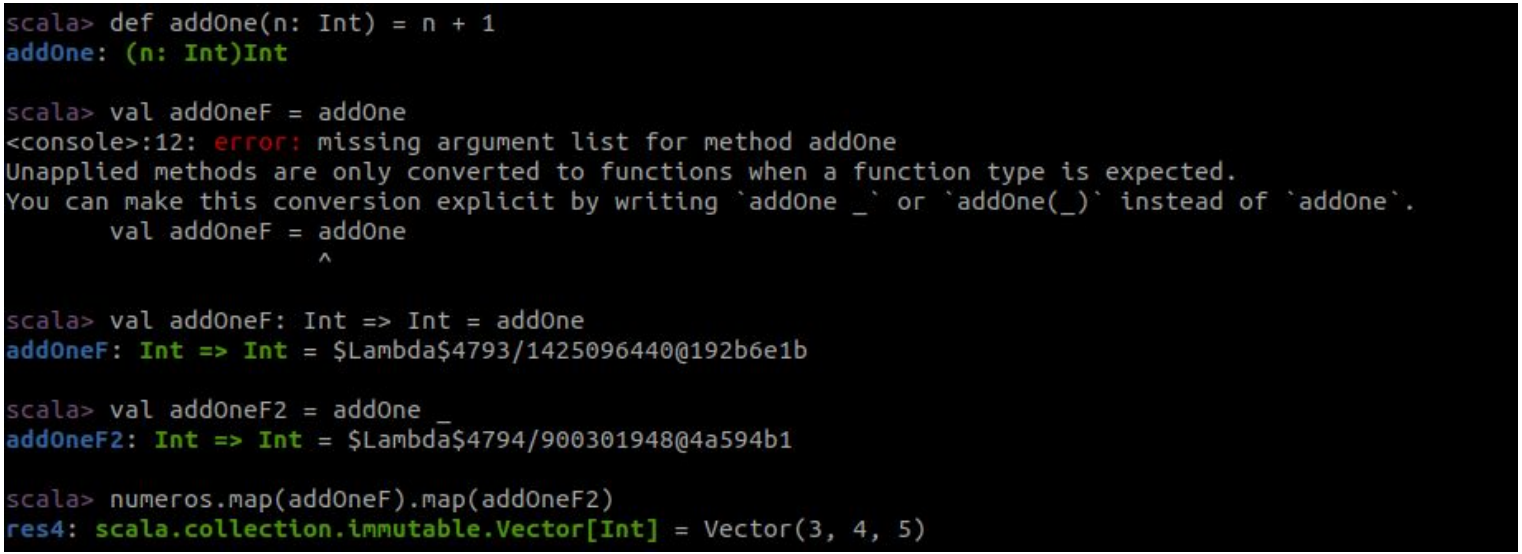
- Los tipos son objetos, pero… ¿cuál es su tipo?
- Desde el REPL vemos la versión endulzada (azúcar sintáctico):
  - *Int => Int* equivalente a *Function1[Int, Int]*

```
scala> val sumaUno = (n: Int) => n + 1
sumaUno: Int => Int = $Lambda$4646/1109386584@3327b00a
```

- La librería estándar tenemos desde *Function0* a *Function22*.
- Todos los tipos de las funciones definen el método *apply* (usado para invocarlas)

```
scala> sumaUno(3)
res3: Int = 4
```

Métodos como Funciones



- Los métodos y las funciones similares, pero diferentes.
- Un método se promueve a función estableciendo el tipo función al valor (variable) que almacenará la referencia.

3 Funciones de orden superior destacadas

Función de orden superior: map


```
scala> val s = Vector("Scala", "Python", "R")
s: scala.collection.immutable.Vector[String] = Vector(Scala, Python, R)

scala> s.map(lenguaje => lenguaje.toLowerCase)
res5: scala.collection.immutable.Vector[String] = Vector(scala, python, r)

scala> s.map(lenguaje => lenguaje.length)
res6: scala.collection.immutable.Vector[Int] = Vector(5, 6, 1)
```

- *map* transforma una colección tras la aplicación de una función a cada elemento de la colección que la invoca.
- El tipo de la colección se mantiene, pero el tipo de los elementos pueden cambiar.

Ejercicio: Usa la función map

- Adapta el tipo de `Train.schedule` a un valor inmutable de tipo `Seq[(Time, Station)]`

- Añade un atributo a `Train` llamado `stations`
  - el tipo de `stations` será `Seq` de `Station`
  - se inicializa con todas las `Station`s que estén en `schedule`

Función de orden superior: flatMap


```
scala> Seq("Bye people", "I will be back").map(s => s.split(" "))
res7: Seq[Array[String]] = List(Array(Bye, people), Array(I, will, be, back))

scala> Seq("Bye people", "I will be back").flatMap(s => s.split(" "))
res8: Seq[String] = List(Bye, people, I, will, be, back)
```

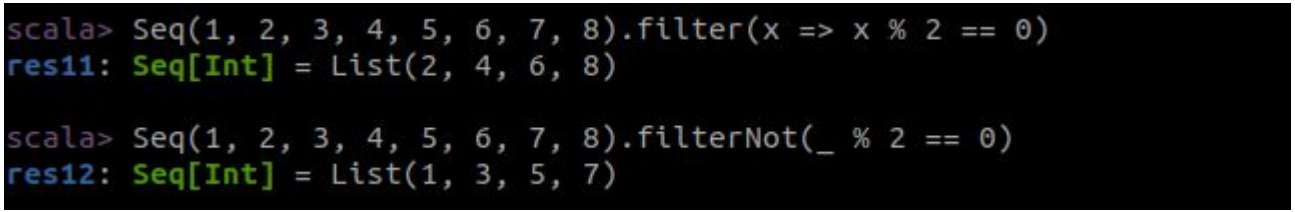
- flatMap, como map, transforma una colección aplicando una función a cada elemento.
- Cada colección para cada elemento es volcado en la colección resultado.

Ejercicio: Usa la función flatMap

- Crea la clase `JourneyPlanner`
  - añade `trains` como parámetro de clase de tipo `Set` de `Train`

- Añade el atributo inmutable `stations` a `JourneyPlanner`
  - inicializa el atributo con todas las `Station`s de todos los `Trains`
  - ¿qué tipo tendría sentido?

Función de orden superior: filter



- Espera una función *predicado* (condición) de un argumento como argumento que devuelva un valor *Boolean*.
- *filter* genera una colección con aquellos elementos seleccionados que satisfacen el predicado.
- *filterNot* genera una colección con los elementos que no satisfacen el predicado.

Ejercicio: Usa la función filter

- Añade el método `trainsAt` a `JourneyPlanner` que tome como parámetro `station` de tipo `Station`:
    - debe devolver todos los `Train` que contenga el valor de `station` en su atributo `stations`
    - ¿qué tipo tiene sentido que tenga?