

CURSO DE PROGRAMACION SCALA

Sesión 4

Sergio Couto Catoira
ingscc00@gmail.com

Índice

- › Object y companion object
- › Case classes
- › Excepciones
- › Trait
- › Sobreescritura de métodos

Object

- > Similar a clases singleton
- > La clase main de nuestra APP se crea como object

```
* Created by scouto.  
*/  
object MyApp extends App {  
    println("Hello World ")  
}
```

Ejercicio

- > Define el companion object para las clases Alumno y Asignatura de la sesión anterior
- > Implementa sus métodos apply y unapply
- > Define una función cualquiera que aplique pattern matching sobre Alumno y otra que lo aplique sobre asignatura

Case classes

- > Clases que implementan automáticamente su companion object con sus métodos apply y unapply
- > Aportan también un toString por defecto más legible
- > Sencillez, evitar escribir de más etc..
- > Atributos val por defecto (se pueden leer, pero son inmutables)
- > Para modificarlos, se emplea el método copy indicándole de 0 a n parámetros
 - p.copy()
 - p.copy(nombre = "other")
- > Se genera también el método equals para la comparación
- > No tenemos que escribir explícitamente los métodos => evitar errores

Clase + companion object

```
class Persona (val nombre: String, val apellidos: String){  
    override def toString: String = nombre + " " + apellidos  
}  
  
object Persona{  
    def apply (nombre: String, apellidos: String): Persona = new Persona(nombre, apellidos)  
    def unapply (persona: Persona): Option[(String, String)] = {  
        Some((persona.nombre, persona.apellidos))  
    }  
}
```

Case class

```
case class Persona(nombre: String, apellidos: String)
```

Ejercicio

- > Redefine las clases Alumno y Asignatura como case classes
- > Comprueba que los métodos creados en esta y la anterior sesión siguen funcionando

Excepciones

- > Se consideran efectos de lado, por lo que deben controlarse dentro de la propia función.
- > Se lanzan igual que en java:
 - `def div (x: Double, y: Double) = if (y != 0.0) x/y else throw new ArithmeticException("denominador debe ser distinto de 0")`
- > El método no declara qué excepciones devuelve.

Capturando Excepciones

- > Try-catch-finally idéntico a java
- > Objeto Try de paquete scala.utils, devuelve Success o Failure
 - Try(q.toInt).isSuccess
 - Try(q.toInt).get
 - Try(q.toInt).getOrElse
- > Puede usarse con pattern matching

Ejercicio

- > Define una función que elimine de una lista el primer elemento que satisfaga una función
 - `def removeFirstElement(list: List[Int], f: Int => Boolean): List[Int]`
- > La función anterior sólo vale para un tipo de datos (Int), generalízala para cualquier tipo de datos.

Traits

- > Similares a interfaces en Java
- > A diferencia de las clases abstractas, no pueden llevar parámetros
- > Una clase sólo puede heredar de una clase abstracta, pero podría heredar de n traits (mixin)
- > Pueden tanto definir como implementar métodos
- > Se usan en subtyping y pueden aplicarse pattern matching si las clases que las implementan se definen como case classes

Ejercicio

- > Reproduce el ejercicio de la sesión 3, definiendo dos tipos de alumno:
 - AlumnoRepetidor
 - AlumnoNuevo
 - Ambos heredarán de Alumno
- > Haz lo mismo con las asignaturas
 - AsignaturaConPrioridad
 - AsignaturaSinPrioridad
 - Ambas heredarán de Asignatura
- > Se comportarán de la siguiente forma:
 - Baja: Igual
 - Alta en Asignatura sin prioridad: Igual
 - Alta en Asignatura con prioridad: Tienen prioridad los alumnos nuevos. Si se da de alta un nuevo y no hay plazas, debe expulsarse a un repetidor de la asignatura. Pista: Deberás usar una función definida anteriormente en clase

Ejercicio

- > Escribir una función recursiva que devuelva el valor fibonacci, la función debe ser tail-safe
 - `def fib(x: Int): BigInt`
- > Escribe un test unitario para ella
- > Escribe una función mergeSort en scala y una función isSorted. Úsala en un test unitario
 - `def msort[T](l: List[T], less(T, T) => Boolean): List[T]`
 - `def isSorted[A](as: List[A])(implicit ordered: (A,A) => Boolean): Boolean`