Práctica 10: POO en Scala (1)

Práctica de Objetos Funcionales en Scala

El capítulo 6 del libro de Martin Odersky *Programming in Scala* (enlace) explica en profundidad usando el ejemplo de los números racionales el concepto de *objeto valor* o *objeto funcional* (son términos equivalentes). En el capítulo se explica cómo implementar este tipo de objetos, usando características de Scala. Muchas de las características ya las hemos visto en teoría y el capítulo te puede servir para repasarlas:

- Caja “Immutable object trade-offs” (pag. 97)
- 6.5: Adding fields (pag. 99)
- 6.6: Self references (pag. 101)
- 6.7: Auxiliary constructors (pag. 102)
- 6.8: Private fields and methods (pag. 104)
- 6.9: Defining operators (pag. 105)
- 6.10: Identifiers in Scala (pag. 107)
- 6.11: Method overloading (pag. 110)

Otras características son nuevas, y debes leer el apartado correspondiente para entenderlas:

- 6.3: Reimplementing the `toString` method (pag. 98)
- 6.4: Checking preconditions (pag. 99)
- 6.12: Implicit (pag. 112)

Ejercicio 1

Implementa la clase `Rational` tal y como se explica en el capítulo 6 de *Odersky* (pag. 111).

Prueba **todos los métodos** de la clase usando `asserts`.

Ejercicio 2

Define, implementa y prueba la clase `Intervalo` como *objeto funcional*, que represente un intervalo en la recta real con su punto de inicio y su punto de fin (defínelos como Double). Prueba **todos los métodos de la clase** usando `asserts` con valores distintos de los que hemos usado como ejemplo.

Define los siguientes constructores:

- `(inicio: Double, fin: Double)` : constructor por defecto. Debe cumplirse (*require*) que `inicio <= fin` . **Importante: no uses la palabra `final` como nombre del atributo que guarda el punto de finalización, porque es una palabra**

reservada de Scala.

- `(posicion: Double)` : devuelve un *intervalo puntual*, el intervalo con los puntos de inicio y de finalización igual al número que se pasa como parámetro.
- `(otro: Intervalo)` : constructor auxiliar que construye un intervalo a partir de otro.

Define los siguientes operadores y métodos:

- `toString` : para mostrar un intervalo con el formato `[3.0, 4.3]` .
- `+(otro: Intervalo): Intervalo` : devuelve un nuevo intervalo que engloba a los dos.
  Ejemplo: `[2.0, 3.0] + [5.3, 6.1] => [2.0, 6.1]`
- `+(desp: Double): Intervalo` : suma el `desp` a la posición inicial y final del intervalo.
  Ejemplo: `[2.0, 3.0] + 1.5 => [3.5, 4.5]`
- `amplia(tamaño: Double): Intervalo` : amplía el intervalo una cantidad. Resta el `tamaño` a la posición inicial y lo suma a la posición final.
  Ejemplo: `[2.0, 3.0] amplia 0.5 => [1.5, 3.5]`
- `reduce(tamaño: Dobule): Intervalo` : reduce el intervalo una cantidad. Suma el `tamaño` a la posición inicial y lo resta a la posición final.
  Ejemplo: `[2.0, 3.0] reduce 0.3 => [2.3, 2.7]` . Si la reducción “hace desaparecer” el intervalo se debe devolver un intervalo puntual situado en el punto medio de la posición inicial y final.
  Ejemplo: `[2.0, 3.0] reduce 0.6 => [2.5, 2.5]`
- `intersecta(otro: Intervalo): Boolean` ,
  `engloba(otro: Intervalo): Boolean` : comprueba si *self* intersecta o engloba el intervalo que se pasa como parámetro.
  Ejemplo `[2.5, 6.2] engloba [2.5, 3.0] => true` ,
  `[2.5, 6.2] engloba [1.0, 6.5] => false` ,
  `[2.5, 6.2] intersecta [1.0, 6.5] => true`
- `interseccion(otro: Intervalo): Interseccion` : devuelve la intersección de los intervalos o `null` si no intersectan.
  Ejemplo: `[2.5, 6.2] interseccion [1.0, 6.5] => [2.5, 6.2]` ,
  `[2.5, 6.2] interseccion [4.0, 10.0] => [4.0. 6.2]` ,
  `[2.5, 6.2] interseccion [10.0, 11.0] => null` .

Ejercicio 3

Implementa la clase árbol `Tree` de tipo `Int` como un objeto funcional con los mismos atributos que vimos en Scheme (*dato* y *listaHijos*) y con los siguientes métodos:

- `toList(): List[Int]` ⇒ devuelve una lista con los enteros del árbol en pre-orden (primero el dato de la raíz y después los hijos, en el orden definido por la lista)
- `numNodos: Int` ⇒ devuelve el número de nodos del árbol
- `sumTotal: Int` ⇒ devuelva la suma de todos sus nodos

Prueba **todos los métodos** de la clase usando `asserts`.
