Ejercicio

Implementa en Scala la función `cuentaIguales` que recibe una lista de tuplas de dos enteros y una función f y devuelve el número de elementos de la lista que cumplen que la aplicación de f sobre el primer elemento de la tupla devuelve el segundo.

```
cuentaIguales(List[(1,1),(2,3),(2,4),(5,6),(5,25)], (x)=>{x*x}) => 2
```

Implementa 3 versiones:

a) (0,5 puntos) Recursiva pura
b) (0,5 puntos) Recursión por la cola
c) (0,5 puntos) Con funciones de orden superior

Ejercicio

a) (1 punto) Utilizando las características de Scala que agilizan la definición de clases en POO, define e implementa la clase `TernaryTree` como un objeto funcional, que define un árbol que contiene un dato entero y 3 hijos que son árboles ternarios, tal y como indican los ejemplos:

```
val t1 = new NodoTernaryTree(2,Vacio,Vacio,Vacio)
val t2 = new NodoTernaryTree(5,t1,Vacio,Vacio)
val t3 = new NodoTernaryTree(10,Vacio,Vacio,Vacio)
val t4 = new NodoTernaryTree(1,Vacio,Vacio,Vacio)
val t5 = new NodoTernaryTree(4,Vacio,t4,Vacio)
val tree = new NodoTernaryTree(12,t2,t3,t5)
```

b) (1 punto) Añade a la clase anterior el método `sumaNodos` , que sume los nodos del árbol:

```
t1.sumaNodos() // ⇒ 2
t5.left.sumaNodos() // ⇒ 0
t5.mid.sumaNodos() //⇒ 1
t5.right.sumaNodos() // ⇒ 0
tree.sumaNodos() // ⇒ 34
```

c) (1 punto) Modifica y/o añade el código necesario para que sólo se puedan crear instancias de `TernaryTree` sin utilizar `new`. Además, la clase `TernaryTree` debe proporcionar el número total de objetos `TernaryTree` que se han creado.

```
val t1 = new NodoTernaryTree(2,Vacio,Vacio,Vacio)  // ⇒ error
```

```
val t1 = NodoTernaryTree(2,Vacio,Vacio,Vacio)
val t2 = NodoTernaryTree(5,t1,Vacio,Vacio)
val t3 = NodoTernaryTree(10,Vacio,Vacio,Vacio)

NodoTernaryTree.numTotalTrees() // ⇒ 3
```

Ejercicio

Supongamos que queremos codificar un juego de rol en el que aparecen distintos tipos de personas, como magos o guerreros, con las siguientes características:

- Las personas tienen un nombre que no cambia y una puntuación que puede ir modificándose con el tiempo.
- La puntuación es una tupla de 3 elementos que representan los valores de poder, resistencia y velocidad.
- Dependiendo del tipo de persona, los valores iniciales de la puntuación son los siguientes:

Tipo de persona	**Puntos iniciales**
Mago	Poder = 5, Resistencia = 2, Velocidad= 2
Guerrero	Poder = 3, Resistencia = 4, Velocidad= 4

- La puntuación de una persona se puede aumentar añadiéndole una cantidad a alguna de sus características.

Lee detalladamente el siguiente ejemplo de código en el que mostramos el funcionamiento de estas clases:

```
val gandalf = new Mago("Gandalf")
val aragorn = new Guerrero("Aragorn")
val persona = new Persona("Sin nombre")
<console>:8: error: class Persona is abstract; cannot be instantiated
val persona = new Persona("Sin nombre") { var puntos = (0,0,0) }
persona.saluda // -> java.lang.String = Hola, soy Sin nombre
gandalf.saluda // -> java.lang.String = Hola, soy el mago Gandalf
gandalf.puntos // -> (Int, Int, Int) = (5,2,2)
gandalf.aumentaPuntos("Velocidad",2)
gandalf.puntos // -> (Int, Int, Int) = (5,2,4)
```

a) Codifica en Scala las clases `Persona`, `Mago` y `Guerrero`, sus atributos y sus métodos para que cumplan las características anteriores.

b) Define en Scala la función `sumaPuntos` que suma todos los puntos de un equipo de personas. Su perfil es el siguiente:

```
// Devuelve una tupla con la suma de todos los puntos de un equipo
def sumaPuntos(equipo: List[Persona]) : (Int,Int,Int)
```

Importante: haz una implementación *no imperativa*. Puedes hacerla recursiva (llamándose a si misma) o implementarla con alguna llamada a alguna función de orden superior de Scala.

c) Crea una nueva clase `Puntuacion` que represente una puntuación de poder, resistencia y velocidad. Explica cómo tendrías que cambiar la clase `Persona` y sus derivadas para que trabajaran con esta nueva clase en lugar de con tuplas.

Escribe una nueva implementación de la función `sumaPuntos`, ahora con este nuevo perfil:

```
// Devuelve la puntuación con la suma de todos los puntos de un equipo
def sumaPuntos(equipo: List[Persona]) : Puntuacion
```

Ejercicio

a) (0,5 puntos) Señala y corrige los 5 errores que hay en el siguiente código:

```
class IntQueue {
  def put(x:Int): Unit
}

class BasicIntQueue extends IntQueue {
  private val buf = new ListBuffer[Int]
  override def put(x) = buf += x
}

trait GetDouble with IntQueue {
  abstract override def get() = 2 * super.get()
}

class MiCola with BasicIntQueue with GetDouble
```

b) (1 punto) Dada la siguiente jerarquía de clases:

```
abstract class Clase1 {
   def g(x:Int,y:Int): Int
   def f(x:Int) = x*2
}
```

```
trait Trait1 extends Clase1{
   abstract override def g(x:Int, y:Int):Int = super.g(x+2,y+2)
}

class Clase2 extends Clase1 {
   def g(x:Int, y:Int) = x+y
   override def f(x:Int) = x+5
}

class Clase3 extends Clase1 {
   def g(x:Int,y:Int) = x+y+20
}

trait Trait2 extends Clase3 {
   def h(x:Int, y:Int) = super.g(x,y+10) + super.f(x+10)
}
```

b.1) Indica las instrucciones que produzcan error y por qué se produce:

```
val a = new Clase1 with Trait1
val b = new Clase3 with Trait2
val c = new Clase3 with Trait2 with Trait1
val d = new Clase2 with Trait1 with Trait2
val e: Clase1 = new Clase3 with Trait2
```

b.2) Indica el valor que devuelven las siguientes instrucciones (tacha las que produzcan error):

```
a.g(2,3)
b.g(3,1)
c.g(1,2)
c.h(1,1)
d.f(4)
d.g(2,4)
d.h(1,1)
e.h(2,2)
```