Nuevas Tecnologías de la Programación

Tema 2: visión preliminar de Scala

Curso 2016-17

Índice

Introducción

Características básicas de Scala:

- escalable: diseñado para poder crecer en función de las necesidades de los usuarios
- fácil de usar y aprender: basado en la plataforma Java
- paradigma: **paradigma funcional+orientación a objetos**

Programación funcional no pura, orientación a objetos pura

Introducción

Más características:

1. conciso
2. alto nivel: facilita el uso de abstracción
3. uso de tipos estáticos: tipo de variables y expresiones deducido en función de los valores que almacenan y manejan. Ventajas:
   - verificable
   - fácil de refactorizar

Índice

Instalación de Scala

Scala es un lenguaje que se ejecuta en la máquina virtual de **Java**, de forma que precisa su instalación para usarlo. Se recomienda instalar la última versión disponible (conviene instalar el kit completo de desarrollo **jdk** y no sólo el entorno de ejecución **jre**).

Asumimos trabajo en **linux** y que ya se dispone del **jdk** de **Java**.

Instalación de Scala

Una vez disponible **Java** se procede a la instalación de **Scala**. La página base para este lenguaje es `http://www.scala-lang.org`. La versión actual (a día 16 de febrero de 2018) es la **2.12.4.**

Hay enlaces específicos para la instalación de Scala, junto a un entorno integrado de desarrollo (que recomiendo) llamado **IntelliJ** o mediante **sbt** (Scala build tool).

Instalación de Scala

La instalación del IDE supone la instalación de las herramientas de **Scala** sólo para uso de dicho entorno de trabajo. Aunque conviene instalar **IntelliJ**, se recomienda también disponer de **sbt**, para poder trabajar desde línea de comandos.

Instalación de Scala

Si la instalación de **sbt** se hizo de forma correcta, al teclear **scala** en la línea de comandos debería obtenerse un mensaje similar al siguiente:

```
mgomez@poldo:~> scala
Welcome to Scala 2.12.1 (Java HotSpot(TM) 64-Bit Server VM,
                                                    Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.

scala>
```

Este comando produce el inicio de la herramienta conocida como **REPL** (read - evaluate - print - loop).

Índice

Intérprete de Scala

Objetivo: familiarizarse con su uso para ir introduciendo los conceptos básicos del lenguaje

```
scala> 1+7
res0: Int = 8
```

Todo lo que es un resultado lo mete en resX

```
scala> res0*3
res1: Int = 24

scala> println("Hola, mundo")
Hola, mundo
```

Se pueden introducir sentencias sin necesidad de estar dentro de una clase

Índice

Definición de variables

Dos tipos básicos de variables: **var** y **val** (variables usuales y variables cuyo valor no puede modificarse)

Identifica el tipo de la variable en funcion de la asignacion

```
scala> val mensaje="Hola, mundo"
mensaje: String = Hola, mundo

scala> mensaje="Adios, mundo"
<console>:8: error: reassignment to val
       mensaje="Adios, mundo"
               ^
```

El contenido NO SE PUEDE CAMBIAR una vez ha sido inicializada

Definición de variables

Puede usarse **var** para cambiar el valor:

Para cadenas, comillas dobles.

```
scala> var saludo="Hola, mundo"
saludo: String = Hola, mundo

scala> saludo="Adios, mundo"
saludo: String = Adios, mundo
```

Índice

Definición de funciones

Palabra reservada def + <nombre_funcion>+(arg1:tipo1, ...): tipo_funcion

Sintaxis:

```
scala> def max(x: Int, y:Int):Int = {
     | if(x > y) x
     | else y
     | }
max: (x: Int, y: Int)Int
```

Aqui el if devuelve x o y: por tanto podría asignar a val resul = al resultado de esta estructura de control

Definición de funciones

En este caso se podría haber escrito de forma más sencilla:

```
scala> def max(x: Int, y: Int) = if(x > y) x else y
max: (x: Int, y: Int)Int
```

En las funciones recursivas siempre hay que especificar el tipo de retorno

Definición de funciones

Y la forma de uso es la habitual:

```
scala> max(3,5)
res3: Int = 5
```

Definición de funciones

def mostrarSaludo = “Hola mundo”
es lo mismo que
val saludo = “Hola mundo”

Para el caso de funciones sin argumento y que no devuelven nada:

```
scala> def mostrarSaludo = println("Hola, mundo")
mostrarSaludo: Unit
```

Unit es el equivalente a “void”. Aqui si se devuelve algo que es del tipo Unit

La salida del intérprete de escala se hace con el comando **:quit** o simplemente **:q**

Aqui todo es un objeto, todo es una clase

Índice

Scripts de Scala

La creación de **scripts** es sencilla: basta con escribir algunas sentencias en un archivo con extensión **scala**. Por ejemplo, supongamos que en el archivo **hola.scala** almacenamos la siguiente sentencia:

```
println("Hola, mundo")
```

Para ejecutarlo bastaría hacer (en el directorio donde está el **script**)

```
scala hola.scala
```

Scripts de Scala

También es posible pasar argumentos al **script**. Supongamos que en el archivo **holaArgumentos.scala** escribo la sentencia:

```
println("Hola, "+args(0))
```

Al ejecutar podría hacer:

```
scala hola.scala Pepe
```

Índice

Estructuras básicas de control

Ejemplo de **while**: Esta no la deberiamos usar

```
var i=0

while(i < args.length){
   println(args(i))
   i=i+1
}
```

Estructuras básicas de control

Usando iteración interna mediante **foreach**:

```
args.foreach(arg => println(arg))
```

De forma mucho más concisa:

```
args.foreach(println)
```

Referencia funcional aqui es la funcion en si

Estructuras básicas de control

Expresión **for**: Esta version no produce resultado

```
for(arg <- args)
   println(arg)
```

Iterador: arg

Índice

Parametrización de arrays con tipos

Ejemplo: array de **String**

```
scala> val saludos=new Array[String](3)
saludos: Array[String] = Array(null, null, null)

scala> saludos(0)="Hola"

scala> saludos(1)=", "
```

Parametrización de arrays con tipos

Usando la expresión **for**:

```
scala> for(i <- 0 to 2)
     | println(saludos(i))
Hola
,
mundo
```

Parametrización de arrays con tipos

Notas:

- forma de interpretación de **0 to 2**: $(0).to(2)$
- todos los operadores trabajando sobre objetos
- acceso a posiciones del array con índices, pero usando paréntesis en lugar de corchetes
- acceso mediante método **sapply**
- asignación de valor mediante método **update**
- forma habitual de inicialización: al crear el array

Índice

Listas

Como esta garantizada la inmutabilidad, se pueden utilizar punteros para reutilizar los objetos. Esto permite que no se tenga que utilizar sincronizacion cuando se necesita paralelismo

Listas como colección inmutable: las operaciones sobre ellas generan nuevas listas (se refuerzan así los principios básicos de programación funcional, con las ventajas derivadas de ello). Todos los elementos de la lista deben ser del mismo tipo

Listas

Ejemplos de creación de listas:

```
scala> val lista1=List(1,2,3)
lista1: List[Int] = List(1, 2, 3)

scala> val lista2=List(4,5)
lista2: List[Int] = List(4, 5)

scala> val lista12=lista1 ::: lista2
lista12: List[Int] = List(1, 2, 3, 4, 5)
```

Listas

Operación habitual: agregar elemento al principio

```
scala> val lista3=1::lista2
lista3: List[Int] = List(1, 4, 5)
```

Los operadores que finalizan en : se invocan sobre el objeto que aparece a la derecha

Nil se usa para denotar la lista vacía

Al ser la lista inmutable, ¿hay que copiar los valores de lista2 en lista3?

Listas

El conjunto de operaciones sobre listas es muy amplio:

- lista1(2): acceso a elemento
- count: conteo de elementos que cumplen una determinada condición

```
scala> lista1.count(s => s > 2)
res2: Int = 1
```

- drop: elimina el número de elementos pasado como argumento

```
scala> lista1.drop(1)
res3: List[Int] = List(2, 3)
```

- dropRight: elimina elementos del final de la lista

```
scala> lista1.dropRight(1)
res4: List[Int] = List(1, 2)
```

Listas

Operaciones sobre listas:

- exists: determina si existe un elemento que cumpla una cierta condición:

```
scala> lista1.exists(s => s==1)
res6: Boolean = true
```

- length: determina la longitud de la lista

```
scala> lista1.length()
<console>:9: error: Int does not take parameters
              lista1.length()
                           ^


scala> lista1.length
res10: Int = 3
```

- head: obtiene el primer elemento de la lista

```
scala> lista1.head
res13: Int = 1
```

Listas

Más operaciones sobre listas:

- foreach: iteración sobre los elementos

```
scala> lista1.foreach(println)
1
2
3
```

- tail: lista con todos los elementos menos el primero

```
scala> lista1.tail
res19: List[Int] = List(2, 3)
```

Índice

Tuplas

Colección inmutable que puede contener elementos de diferentes tipos

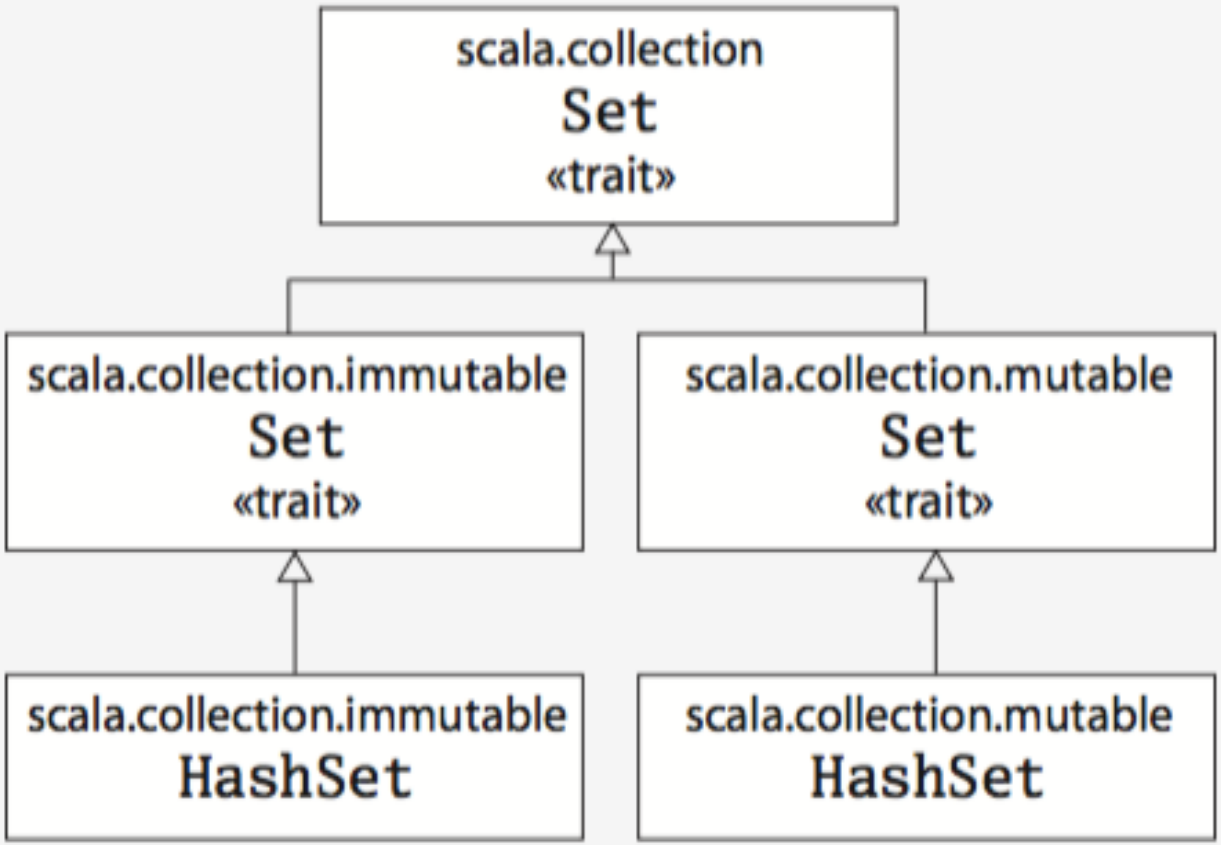
```
scala> val tupla1=(1,"lunes")
tupla1: (Int, String) = (1,lunes)

scala> val tupla2=(1,"lunes","L")
tupla2: (Int, String, String) = (1,lunes,L)

scala> println(tupla1._1)
1

scala> println(tupla1._2)
lunes
```

En las tuplas se empieza a contar desde 1

Índice

Mapas y conjuntos

Scala ofrece versiones mutables e inmutables para estos tipos.



Ejemplo de creación de un conjunto:

```
scala> var ciudades=Set("Granada", "Jaen", "Almeria")
ciudades: scala.collection.immutable.Set[String] =
                    Set(Granada, Jaen, Almeria)
```

Al ser **ciudades** una variable (mutable) es posible agregar elementos:

```
scala> ciudades+="Malaga"
```

En realidad, lo que ocurre es que ciudades termina apuntando a un nuevo objeto de tipo **Set** (ya que el conjunto en sí es inmutable).

NOTA: para los conjuntos, por defecto, se usa la versión inmutable.

Conjuntos y mapas

Para usar la versión mutable hay que importar la clase correspondiente:

```
scala> import scala.collection.mutable.Set
import scala.collection.mutable.Set

scala> val asignaturas=Set("Matematicas", "Fisica")
asignaturas: scala.collection.mutable.Set[String] =
                                Set(Fisica, Matematicas)
```

Al ser mutable la agregación se hace sobre el propio conjunto, sin crear otro nuevo. Por eso puede operarse con **val**

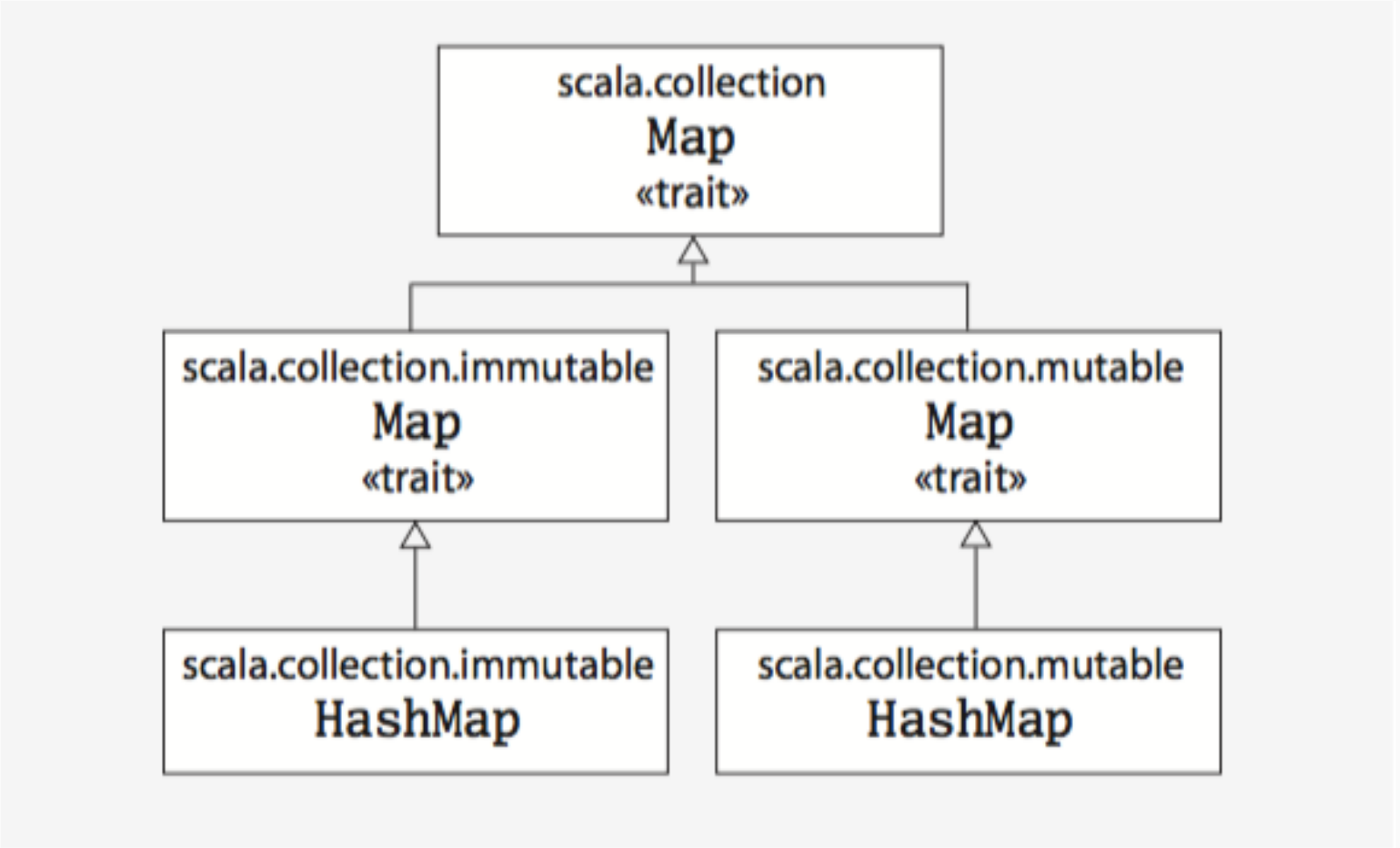
```
scala> asignaturas+="Lengua"
res1: asignaturas.type = Set(Fisica, Lengua, Matematicas)
```

Para usar **HashSet** siempre hay que especificar la clase concreta a usar (mutable o inmutable) mediante la sentencia **import** correspondiente:

```
scala> import scala.collection.immutable.HashSet
import scala.collection.immutable.HashSet

scala> val conjunto=HashSet("Granada", "Almeria")
mapa: scala.collection.immutable.HashSet[String] =
                                        Set(Almeria, Granada)
```

Conjuntos y mapas

La jerarquía de clases para mapas es:



Conjuntos y mapas

Ejemplo de creación de mapas:

```
scala> import scala.collection.mutable.Map
import scala.collection.mutable.Map

scala> val dias=Map[Int,String]()
dias: scala.collection.mutable.Map[Int,String] = Map()

scala> dias+=(1 -> "Lunes")
res1: dias.type = Map(1 -> Lunes)

scala> println(dias(1))
Lunes
```

Ejemplo de creación de mapas (inmutable)

```
cala> val dias=Map[Int, String](1->"Lunes", 2->"Martes")
dias: scala.collection.immutable.Map[Int,String] =
                    Map(1 -> Lunes, 2 -> Martes)

scala> println(dias)
Map(1 -> Lunes, 2 -> Martes)
```

Conjuntos y mapas

Otro ejemplo:

```
scala> val numerosRomanos=Map(1->"I", 2->"II", 5->"V")
numerosRomanos: scala.collection.mutable.Map[Int,String] =
                                Map(2 -> II, 5 -> V, 1 -> I)
```

Índice

Estilo de programación funcional

Comentarios:

- Scala permite desarrollar aplicaciones usando el paradigma imperativo
- los diseñadores del lenguaje comprenden la dificultad para adaptarse al paradigma funcional y que algunas tareas pueden resolverse mejor no siguiendo el paradigma de forma intransigente
- pistas: usar **val** (aunque se permite el uso de **var**) ; usar colecciones inmutables; usar expresiones que devuelvan valor (por ejemplo, en las iteraciones usar for y no while)

Estilo de programación funcional

Ejemplo: impresión de array de argumentos (**argumentos.scala**)

```
def imprimirArgumentos(args : Array[String]) : Unit = {
    var i=0

    while(i < args.length){
        println(args(i))
        i+=1
    }
}
```

Estilo de programación funcional

Forma de ejecución:

```
scala argumentos.scala Hola Pepe adios 3
Hola
Pepe
adios
3
```

Estilo de programación funcional

Versiones mucho más ajustadas al paradigma funcional:

```
def imprimirArgumentos(args : Array[String]) : Unit = {
   for(arg <- args)
      println(arg)
}
```

Estilo de programación funcional

Y mejor:

```
def imprimirArgumentos(args : Array[String]) : Unit = {
    args.foreach(println)
}
```

Estilo de programación funcional

Y presciendiendo de los efectos laterales:

```
def imprimirArgumentos(args : Array[String]) : String = {
   args.mkString("\n")
}

val cadena=imprimirArgumentos(Array("uno","dos","tres"))

assert(cadena == "uno\ndos\ntres")
```

Índice

Ejemplo: lectura de archivo

Consideraremos diferentes versiones de un script para procesar el contenido de un archivo, mostrando la longitud de cada línea y el contenido de la línea en sí. Idealmente la escritura del tamaño de la línea no debería modificar el sangrado del texto.

Ejemplo: lectura de archivo

Primera versión: para cada linea se muestra longitud y contenido:

```
import scala.io.Source

if (args.length > 0){
   for(linea <- Source.fromFile(args(0)).getLines())
      println(linea.length+" "+linea)
}
else
   Console.err.println("Introduzca nombre de archivo")
```

Ejemplo: lectura de archivo

Mantener el sangrado obliga a:

- iterar dos veces sobre las líneas
- en la primera pasada se trata de determinar el número de caracteres máximo para representar la longitud de las líneas
- para facilitar esta doble iteración se almacenan las líneas en una lista

Ejemplo: lectura de archivo

Almacenamiento en lista:

```
val lineas=Source.fromFile(args(0)).getLines().toList
```

Función para determinar el espacio máximo necesario para representar la longitud de las líneas:

```
def calcularAnchoTamLinea(s : String) =
                              s.length.toString.length
```

Ejemplo: lectura de archivo

Al iterar sobre la lista de líneas podemos calcular el máximo ancho necesario para escribir la longitud de la cadena de cada línea. Por ejemplo, si la línea más larga tuviera 123 caracteres, deberíamos obtener 3:

```
var maximoAnchoTam=0
   for(linea <- lineas)
      maximoAnchoTam = maximoAnchoTam.max(
                       calcularAnchoTamLinea(linea))
```

Una vez hecho esto habría que procesar las líneas de nuevo, para mostrar el número de caracteres ajustado a maximoAnchoTam y el contenido de la línea en sí.

Ejemplo: lectura de archivo

```
for(linea <- lineas){
      val tamLinea = calcularAnchoTamLinea(linea)
      val relleno=" "*(maximoAnchoTam-tamLinea)
      println(relleno+linea.length+"|"+linea)
   })
```

Ejemplo: lectura de archivo

Otra aproximación más funcional consiste en obtener la línea más larga (para luego determinar su tamaño y el ancho necesario para representarlo):

```
val lineaMasLarga = lineas.reduceLeft(
       (a,b) => if(a.length > b.length) a else b)
```

Con lo que ahora no es preciso iterar de forma explícita para calcular el máximo ancho del tamaño de la línea:

```
val maximoAnchoTam = calcularAnchoTamLinea(lineaMasLarga)
```

Ejemplo: lectura de archivo

Finalmente, se muestra el contenido debidamente formateado:

```
    for(linea <- lineas){
        // Se calcula el numero de espacios en blanco para
        // igualar con el maximo tamanno de ancho
        val numeroEspacios=maximoAnchoTam-
                           calcularAnchoTamLinea(linea)

        // Se construye la cadena de blancos de relleno
        val relleno=" "*numeroEspacios

        // Se muestra la linea
        println(relleno + linea.length + "|" + linea)
    }
```

Ejemplo: lectura de archivo

```
1  import scala.io.Source
2
3  def calcularAnchoTamLinea(s : String) = s.length.toString.length
4
5  if (args.length > 0){
6     val lineas=Source.fromFile(args(0)).getLines().toList
7     val lineaMasLarga = lineas.reduceLeft(
8         (a,b) => if(a.length > b.length) a else b)
9
10    val maximoAnchoTam = calcularAnchoTamLinea(lineaMasLarga)
11
12    for(linea <- lineas){
13       // Se calcula el numero de espacios en blanco para igualar
14       // con el maximo tamaño de ancho
15       val numeroEspacios=maximoAnchoTam-calcularAnchoTamLinea(linea)
16
17       // Se construye la cadena de blancos de relleno
18       val relleno=" "*numeroEspacios
19
20       // Se muestra la linea
21       println(relleno + linea.length + "|" + linea)
22    }
23 }
24 else
25    Console.err.println("Introduzca nombre de archivo")
```