



Departamento de Informática
Universidad de Valladolid
Campus de Segovia

Tema 3

Sintáxis y elementos básicos de Scala



Contenido

- Objetivos
- Sintaxis básica
- Métodos en Scala
- Entrada y Salida en Scala
- Declaraciones perezosas
- Strings
- Tipo Option
- Manejo de excepciones

Objetivos

- Manejar la sintáxis y semántica básica del lenguaje Scala para poder empezar a implementar programas sencillos.

Sintáxis Básica

Scala, al igual que Java:

- Distingue mayúsculas de minúsculas.
- El nombre del fichero que contiene a la clase no tiene por que coincidir con el nombre de dicha clase, pero se recomienda que coincidan.
- Se sigue el mismo criterio a la hora de identificar clases, variables y métodos:
 - Las nombres de las clases comienzan siempre por mayúsculas
 - Las nombres de las variables y métodos comienzan siempre por minúsculas.
 - En caso de que el identificador propuesto sea compuesto, la primera letra de cada palabra interior debe aparecer en mayúscula.

Sintaxis Básica: Identificadores

- Los identificadores son cadenas alfanuméricas que pueden comenzar con una letra o un guion bajo, y pueden contener letras, dígitos o guiones bajos.
- Ejemplos:
 - edad
 - _valor
 - salario_1

Sintaxis Básica:

Tipos de datos

- En Scala todos los tipos de datos son objetos, incluidas las funciones.

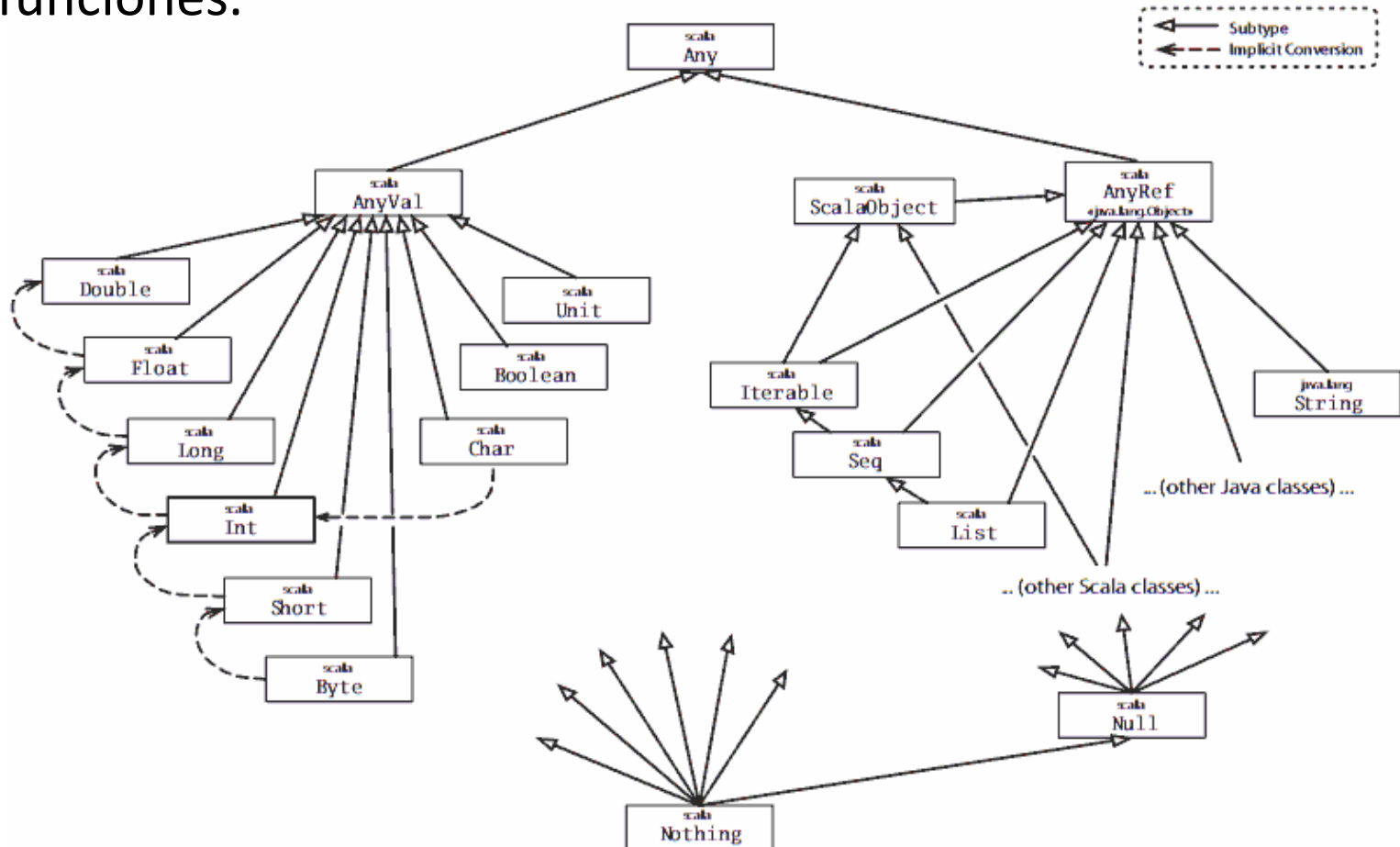


Figure 11.1 · Class hierarchy of Scala.

Sintaxis Básica:

Tipos de Datos

Byte*

Entero con signo de 8 bits (-128 a 127)

Short*

Entero con signo de 16 bits (-32768 a 32767)

Int*

Entero con signo de 32 bits
(-2147483648 a 2147483647)

Long*

Entero con signo de 64 bits
(-9223372036854775808 a 9223372036854775807)

Float*

real con precisión simple (32 bits IEEE 754)

Double*

real con precisión doble (64 bits IEEE 754)

Char*

Carácter Unicode sin signo de 16 bits.
(U+0000 a U+FFFF)

String*

Cadena de caracteres

Boolean

literales true o false

Unit

Ningún valor (no valor)

Null

null o referencia vacía

Nothing

Subtipo de cualquier otro tipo; incluido el no valor

Any

El supertipo de cualquier tipo; cualquier objeto es del tipo *Any*.

AnyRef

El supertipo de cualquier tipo referencia incluidas las definidas por el usuario

*Los literales se construyen como en java

Sintaxis Básica:

declaración de variables(1)

- Sintaxis:

`val | var identificador : TipoData = [Initial Value]`

`val`: Define una referencia como inmutable (constante)

`var`: Define una referencia como mutable (variable)

- Ejemplos:

`var miVar : Int=5`

`val miVal : String="Hola"`

- Inicialización múltiple mediante tuplas.

`val (miVar1, miVar2,miVar3) = Tuple3(5, "Hola", 2.3)`

Sintaxis Básica:

declaración de variables(2)

- val vs var

```
var miVar : Int=5
```

```
val miVal : String="Hola"
```

```
miVar=6 // permitido
```

```
miVal="Pepe"// error de compilación
```

- Una variable declarada como val no acepta nuevas asignaciones.

Sintaxis Básica:

declaración de variables(3)

- **Muy importante:** En Scala, y siguiendo los canones de la programación funcional, se recomienda encarecidamente el uso de **val** para hacer más seguro el uso de referencias y así evitar los efectos laterales.
- El uso de **var** quedaría relegado a aquellas situaciones donde no quede más remedio.

Sintaxis Básica:

Inferencia de Tipos

- Inferencia de tipos. Scala es capaz de inferir el tipo

```
object Prueba extends App {  
    val x = 1 + 2 * 3  
    val y = x.toString()  
    def sucesor(x: Int) = x + 1  
}
```

- Esta inferencia de tipos falla si el algoritmo es de tipo recursivo. En ese caso es necesario incluir el tipo del dato que se retorna.

Sintaxis Básica:

Keywords (Palabras reservadas)

abstract	case	catch	class
def	do	else	extends
false	final	finally	for
forSome	if	implicit	import
lazy	match	new	null
object	override	package	private
protected	return	sealed	super
this	throw	trait	try
true	type	val	var
while	with	yield	
-	:	=	=>
<-	<:	<%	>:
#	@		

Sintaxis Básica:

Tipos de variables y su accesibilidad

- Campos.
 - Son variables que pertenecen a un objeto.
 - Su accesibilidad depende del modificador prefijo (como en Java).
 - Pueden ser mutables o inmutables.
- Parámetros de los métodos.
 - Son las variables que se utilizan para pasar datos a un método cuando este es invocado.
 - Son accesibles desde dentro del método.
 - Estas variables son siempre inmutables.
- Variables locales
 - Son variables creadas dentro de un método.
 - Son sólo accesibles desde dentro del método
 - Pueden ser mutables o inmutables.

Sintaxis Básica:

Comentarios

- Exactamente igual que en Java.
- Ejemplos:
 - // comentario de una sólo Línea
 - /* comentario de un párrafo
con varias líneas*/

Sintaxis Básica: definición de sentencias

- Para separar sentencias se utiliza el punto y coma:
`val s="hello"; println(s)`
- Si las sentencias están en distintas líneas, el punto y coma se puede obviar (se infiere del contexto).

```
val s="hello"  
println(s)
```


Sintaxis Básica:

Operadores Aritméticos

Operador	Descripción	Expresión
+	Suma	$A + B$
-	resta	$A - B$
*	Multiplicación	$A * B$
/	División entera	B / A
%	Modulo (devuelve el resto de la división entera)	$B \% A$

Sintaxis Básica:

Operadores Relacionales

Operator	Description	Expresión
==	igualdad	(A == B)
!=	diferencia	(A != B)
>	Mayor que	(A > B)
<	Menor que	(A < B)
>=	Mayor o igual que	(A >= B)
<=	Menor o igual que	(A <= B)

Sintaxis Básica:

Operadores Lógicos

Operador	Descripción	Expresión
&&	AND	(A && B)
	OR	(A B)
!	NOT	!(A)

Sintaxis Básica:

Operadores Lógicos

Operador	Descripción (bitwise, bit a bit)	Expresión
&	AND binario	(A & B)
	OR binario	(A B)
^	XOR binario	(A ^ B)
~	Complemento a uno.	(~A)
<<	Desplazamiento binario a la izquierda	A << 3
>>	Desplazamiento binario a la derecha	A >> 3

Sintaxis Básica:

Operador asignación.

Operador	Descripción	Expresión
=	Asignación	$C = A + B$
+=	Suma y asigna	$C += A$ es equivalente a $C = C + A$
-=	Resta y asigna	$C -= A$ es equivalente a $C = C - A$
*=	Multiplica y asigna	$C *= A$ es equivalente a $C = C * A$
/=	Divide y asigna	$C /= A$ es equivalente a $C = C / A$
%=	Calcula el resto y los asigna	$C \% = A$ es equivalente a $C = C \% A$
<<=	Desplaza a la izquierda y asigna	$C <<= 2$ es equivalente a $C = C << 2$
>>=	Desplaza a la derecha y asigna	$C >>= 2$ es equivalente a $C = C >> 2$
&=	Bitwise AND y asignación	$C \&= 2$ es equivalente a $C = C \& 2$
^=	bitwise exclusive OR y asignación	$C \wedge= 2$ es equivalente a $C = C \wedge 2$
=	bitwise inclusive OR y asignación	$C = 2$ es equivalente a $C = C 2$

Sintaxis Básica:

Precedencia de Operadores.

Tipo	Operador
Postfijo	() []
Unario	! ~
Multiplicativo	* / %
Aditivo	+ -
Desplazamiento binario	>> >>> <<
Relacional	> >= < <=
Relacioal	== !=
Bitwise AND	&
Bitwise XOR	^
Bitwise OR	
AND Lógico	&&
OR Lógico	
Asignación	= += -= *= /= %= >>= <<= &= ^= =
Coma	,

Sintaxis Básica:

Estructuras de control

- Scala posee, al igual que Java, las estructuras de control básicas:
 - Selección
 - Iteración
- Como lenguaje Funcional también soporta la recursión como herramienta para la implementación de algoritmos: métodos recursivos

Sintaxis Básica:

Estructuras de control: selección(1)

- Sintáxis:

```
if (expresión_Booleana) {  
    cuerpo  
} else {                // esta rama es opcional  
    cuerpo}
```

- Ejemplo:

```
def main(args: Array[String]) {  
    var x = 30  
    if ( x < 20 )  
        println("menor de 20")  
    else  
        println("mayor de 20")  
}
```


Sintaxis Básica:

Estructuras de control: selección(2)

- Case (pattern matching)
Sintaxis:

```
def f(a:A):B =  
  a match{  
    case opcion1 => b1  
    case opcion2 => b2  
    .....  
    case _ => bn // valor por defecto  
  }  
bi ∈ B con i= 1,2,...,n
```

- Ejemplo

```
object PruebaCase {  
  def prueba(x:Int):String=  
    x%2 match{  
      case 0 => "par"  
      case _ => "impar"  
    }  
  def main(args:Array[String]):Unit={  
    println(prueba(5))  
  }  
}
```

Sintaxis Básica:

Estructuras de control: selección(3)

- Ejemplos

```
object PruebaCase {  
  def SioNo(opcion: Int): String =  
    opcion match {  
      case 1 | 2 | 3 => "si"  
      case 0 => "no"  
      case _ => "error"  
    }  
  def main(args:Array[String]):Unit={  
    println(SioNo(5))  
  }  
}
```

- Ejemplos

```
object PruebaCase extends App {  
  def prueba(x: Any): Any = x match {  
    case 1 => "uno"  
    case "dos" => 2  
    case y: Int => "Entero de Scala"  
    case _ => "otra cosa"  
  }  
  println(prueba("dos"))  
  println(prueba(5))  
  println(prueba("Hola"))  
}
```

Estructuras de control: iteración

- Tipos de bucles:

- while

```
while(expresion_booleana){  
    sentencias  
}
```

- do-while

```
do {  
    sentencias  
} while(expresion_booleana)
```

- for

```
for( x <- rango ){ // x es una variable mutable  
    sentencias  
}
```

Ejemplo:

```
for( a <- 1 to 10)  
    println( a )
```

Sintaxis Básica:

Estructuras de control: el bucle for (1)

- Otra forma:


```
for( a <- 1 until 10)  
  println( a )
```

- Bucles anidados:

```
for( a <- 1 to 10; b <- 1 to 20)  
  println( a+" "+b )
```

Equivalente a:

```
for( a <- 1 to 10)  
  for( b <- 1 to 20)  
    println(a+" "+b)
```



- Con Strings:

```
for( a <- "Hola" )  
  println( a )
```

- Con colecciones de datos (listas).

```
val listaEnteros = List(1,2,3,4,5,6)  
for( a <- listaEnteros )  
  println( a )
```

Sintaxis Básica:

Estructuras de control: el bucle for (2)

- Con filtros:

```
for( a <- listaEnteros if a!=2; if a%2==0 )  
  println( a )
```

- Con definiciones:

```
for( a <- 1 to 3; desde=3-a; b <- desde to 3 )  
  println( a+" "+b )
```

- Con la clausula yield: (los filtros son opcionales)

```
val c= for(a <- 0 to 10 if a!=2; if a%2==0) yield a  
for( b<-c )  
  println(b)
```

Sintaxis Básica:

Estructuras de control: el bucle for (3)

Ejemplos:

```
object Prueba extends App {  
  def numPares(ini: Int, fin: Int): List[Int] =  
    for (x <- List.range(ini, fin) if x % 2 == 0) yield x  
  println(numPares(0, 20))  
}
```

```
object Prueba1 extends App {  
  for (i <- Iterator.range(0, 20); j <- Iterator.range(i, 20) if i + j == 32)  
    println("(" + i + ", " + j + ")")  
}
```

Sintaxis Básica:

Estructuras de control: bucle foreach

- Bucle para iterar sobre una lista de elementos
- Se puede utilizar con tipos como List, Array, ArrayBuffer, Vector, Seq, etc.
- ```
def foreach(f:=>Unit):Unit
 val x = List(1,2,3)
 x.foreach { i=>println(i) }
 // también x.foreach(i=>println(i))
```

# Métodos o funciones en Scala



# Métodos ó Funciones

- En Programación Funcional, los métodos o funciones son el elemento central y representan un conjunto de acciones que puede ser manipuladas como un tipo de dato más.
- En Scala, además, los métodos también representan las operaciones que se pueden realizar sobre los tipos de datos (funciones y procedimientos de la POO ).
- Ahora nos centraremos en la declaración e invocación básica de los métodos.

# Declaración de métodos (1)

- Sintáxis:

```
def identificador ([lista de parametros]) : [tipo devuelto]={
 cuerpo de sentencias
 return expresión
}
```

- Lista de parámetros: es una lista separadas por comas de los parámetros con sus correspondientes tipos.
- La palabra **return** es opcional. La expresión devuelta debe ser del tipo que el método retorna.

- Ejemplo:

```
def sumEnt(a:Int, b:Int) : Int = {
 var sum: Int = 0
 sum = a + b
}
// def sumEnt(a:Int, b:Int) =a+b
```

# Declaración de métodos (2)

- Método que no retorna valor (procedimiento):

```
def identificador ([lista de parametros]) : Unit={
 cuerpo de sentencias
}
```

- Ejemplo:

```
def muestraHola() : Unit = {
 println("Hola")
}
```

# El método main

- Método que lanza la ejecución principal del programa (no retorna valor):

```
def main(args: Array[String]) {

}
```

- Este método se define siempre en un objeto singleton:

```
object Prueba{
 def main(args: Array[String]) {

 }
}
```

# Invocación de métodos(1)

- Invocación estándar (call\_by\_value):
  - identificadorFuncion(lista parametros\_formales:tipo)
- Ejemplo:

```
object Prueba {
 def main(args: Array[String]) {
 println("Returned Value : " + sumaEnt(5,7));
 def sumaEnt(a:Int, b:Int) : Int = {
 var sum:Int = 0
 sum = a + b
 }
 }
}
```

- **Accesibilidad:** igual que en Java, a través de los modificadores prefijos public, protected y private.

# Invocación de métodos(2)

- Invocación estándar (call\_by\_value):
  - instancia.identificadorFuncion(lista de parametros formales)
- Si  $a$  es un objeto y  $b$  un parámetro del método, entonces:  
a.metodo(b)

Una forma abreviada sería:  
a metodo b

# Sobreescritura de métodos

- En Scala se debe utilizar el modificador prefijo **override** a la hora de sobreescribir un método:
- Ejemplo:

```
class Persona {

 override def toString = getClass.getName + "nombre" + nombre

}
```

# Métodos recursivos (1).

- Scala, como lenguaje funcional, soporta la implementación de métodos recursivos.
- Un método recursivo es aquel que se invoca a sí mismo en su cuerpo de definición.
- Existen dos tipos:
  - Recursión básica (stack-based): los valores intermedios se amacenan en una pila.
  - `tail_recursion` (local): los valores intermedios pasan como parámetros de la siguiente invocación



# Métodos recursivos (2).

- Head recursion (recursión básica).

```
def factorial(n: Int): Int = if (n == 0) 1 else n * factorial(n - 1)
```

- Tail\_recursion.

```
def factorial(n: Int): Int = {
 @annotation.tailrec
 def go(n: Int, acc: Int): Int =
 if (n <= 0) acc
 else go(n-1, n*acc)
 go(n, 1)
}
```

# Métodos recursivos (2).

## Head recursion

(fact 3)

(\* 3 (fact 2))

(\* 3 (\* 2 (fact 1)))

(\* 3 (\* 2 (\* 1 (fact 0))))

(\* 3 (\* 2 (\* 1 1)))

(\* 3 (\* 2 1))

(\* 3 2)

6

## Tail recursion

(fact 3)

(go 3 1)

(go 2 3)

(go 1 6)

(go 0 6)

6

# Métodos recursivos (3).

- Head recursion (recursión básica).

```
def fib(n: Int): Int = if (n > 1) fib(n - 1) + fib(n - 2) else n
```

Tail\_recursion

```
def fib(n: Int): Int = {
 @annotation.tailrec
 def go(n: Int, a: Int, b: Int): Int =
 if (n == 0) a
 else if (n==1) b
 else go (n-1, b, a+b)

 go(n,0,1)
}
```

Tail Recursion

```
def fib (n: Int): Int = {
 @annotation.tailrec
 def go(a: Int, b: Int, iter: Int): Int = {
 if (n == iter) a else
 go(b, a + b, iter + 1) }

 go(0, 1, 0)
}
```

# Argumentos con valores por defecto

- Existen métodos que se pueden definir mediante argumentos con valores por defecto:

```
def formato(cad:String, izq:String="{", der:String="}"): String=
 izq+cad+der
val s=formato("hola")
```

- También se pueden incluir otros valores si no se quiere que aparezcan los valores por defecto.

```
val s=formato(izq="[", cad="hola", der="]")
val s=formato("hola", "[", "]")
```

# Métodos variádicos (1)

- Un método variádico es una función de aridad indefinida, es decir, que acepta una cantidad de argumentos variable.

```
def suma(xs:Int*):Int={
 var resultado= 0
 for (x <- xs) resultado+=x
 resultado}
```

.....

```
val s = suma(1,2,3,4))
```

- Si se invoca una función variádica a través de una colección de datos es necesario añadir el operador “splat” al argumento.

```
val s = suma(1 to 4:_*) // :_* operador splat
```

# Métodos variádicos (2)

- Otro ejemplo de invocación variádica en un algoritmo recursivo:

```
def sumaRecursiva(args:Int*):Int={
 if(args.length==0) 0
 else args.head + sumaRecursiva(args.tail:_*)
}
```

# El método `apply`(1).

- El método **`apply`** es un método sobrecargado que permite construir objetos a partir de otros (método factory).
- En Scala el método **`apply`** es una forma de poder trabajar con funciones como si fuesen objetos y viceversa.

```
object Suma2a {
 var y = 2
 def apply (x: Int) = x + y }
```

```
object Aplicar {
 def main(arg:Array[String]):Unit={
 println(Suma2a(5)) // forma abreviada de Suma2a.apply(5)
 }
}
```

# El método apply(2).

- Existen métodos apply predefinidos:
  - String
    - `def apply(n:Int): Char`  
`"Hola"(2)` // forma abreviada de `"Hola".apply(2)`
  - BigInt
    - `def apply(cad:String): Int`  
`BigInt("123")` // forma abreviada de `BigInt.apply("123")`
  - Array
    - `def apply(ns:Int*):Array[Int]` // ns es una coma-lista de enteros  
`Array(1,2,3,4)` // forma abreviada `Array.apply(1,2,3,4)`



# El método equals vs el operador ==

- En Scala el operador == siempre delega sobre el método equals.
  - Usa el método equals (**sobreescrito**) para comparar el contenido de dos objetos.
  - Usa el operador == para comparar, sin importar si las referencias son null.
- Ejemplo:
  - **1 equals 2** devuelve false, ya que lo redirige a Integer.equals(...)
  - **1 == 2** devuelve false, ya que lo redirige a Integer.equals(...)
  - **ref equals ref** devuelve true, a menos que ref sea null entonces arroja un NullPointerException
  - **ref == ref** devuelve true, incluso si ref es null



I/O

# Input/Output. Consola(1)

- Entrada: `import scala.io.StdIn._`
- Métodos para la entrada:
  - `readLine()`: lee un string de la consola
  - `readLine(prompt)`: imprime el prompt y lee un string de la consola
  - `readInt()`: lee un número entero de la consola
  - `readDouble()`: lee un número entero de doble precisión de la consola
  - `readBoolean()`: "yes", "y", "true", y "t" como true, y cualquier otra cosa como false
  - `readChar()`: lee un caracter de la consola
- Salida: `print()`, `println()`

# Input/Output. Consola(2)

- Entrada: import java.util.Scanner

```
object Prueba extends App {
 val sc = new Scanner(System.in);
 println(" introduce un nombre")
 val s = sc.nextLine()
 println(s)
 println(" introduce un numero")
 val i = sc.nextInt();
 println(i)
}
```

# Input/Output. Ficheros

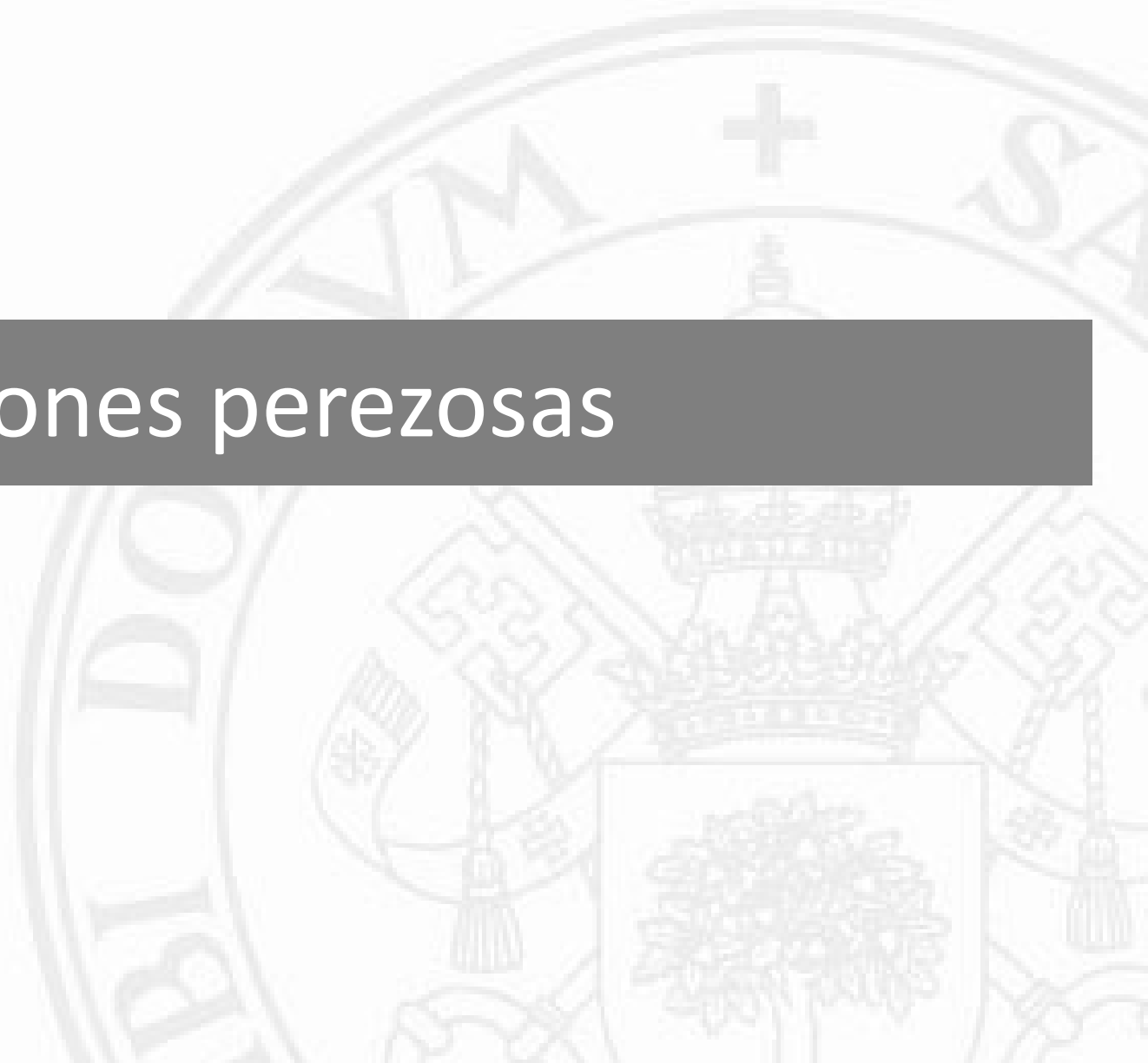
- Se pueden utilizar las clases de Java i/o:

```
import java.io._
object Prueba {
 def main(args: Array[String]) {
 val writer = new PrintWriter(new File("texto.txt"))
 writer.write("Prueba con ficheros")
 writer.close() }
}
```

- Para la lectura se puede emplear la clase Source de Scala:

```
import scala.io.Source
object Prueba {
 def main(args: Array[String]) {
 println("El contenido del fichero:")
 Source.fromFile("texto.txt").foreach {print} }
}
```

# Declaraciones perezosas



# Declaraciones diferidas o perezosas (Lazy values).

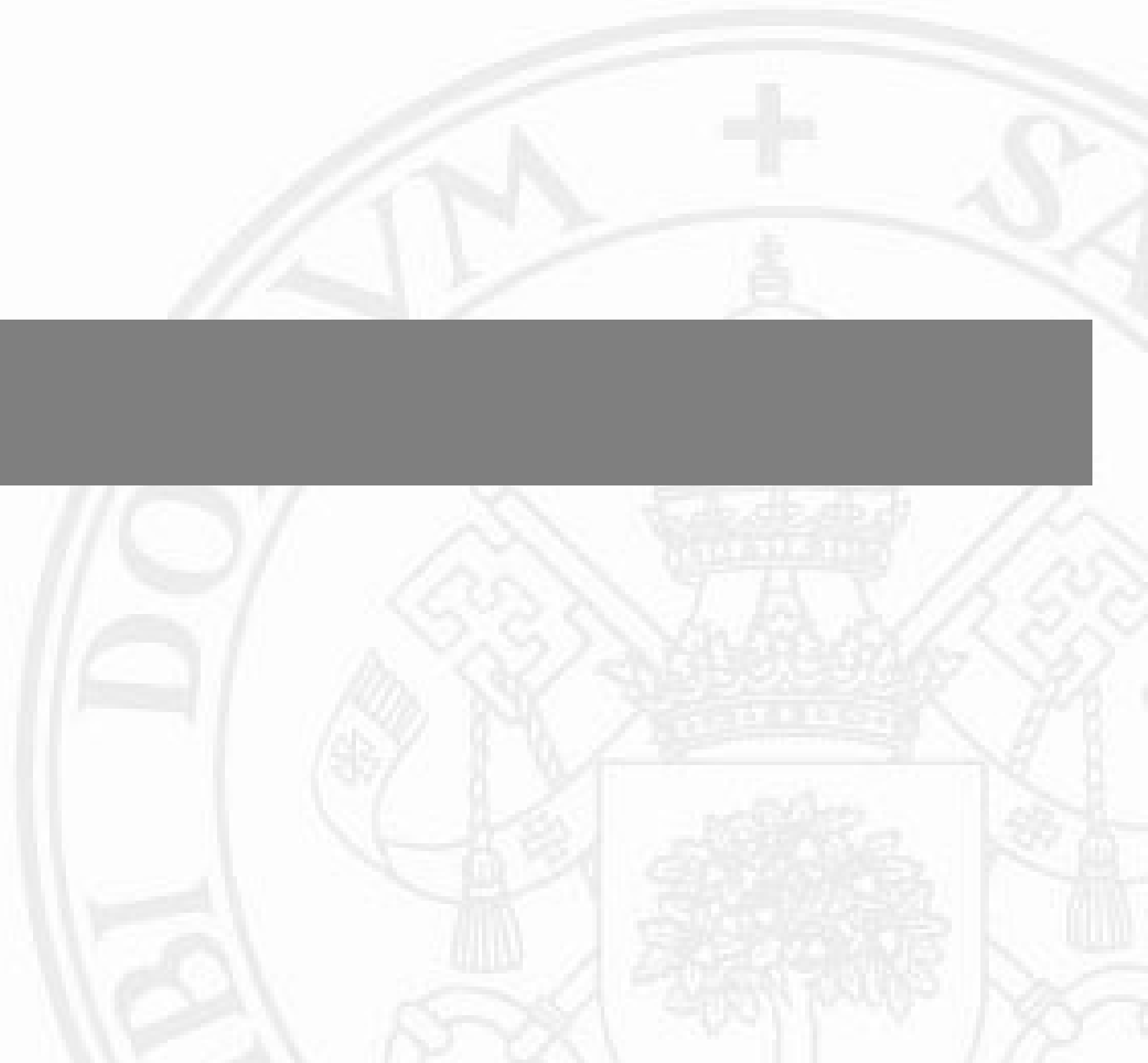
- Cuando una variable inmutable es declarada como *lazy* su inicialización se difiere hasta su primer acceso.

```
lazy val cadena=scala.io.Source.fromFile("../directorio/cad.txt").mkString
//se evalúa la primera vez que se utilice la variable cadena.
//Si cadena no se utiliza nunca el fichero no se abre.
```

- Lazy val vs def

```
def cadena=scala.io.Source.fromFile("../directorio/cad.txt").mkString
//se evalúa cada vez que se invoca la cadena.
//si el fichero cambia, el contenido de cadena cambia.
```

# String





# Strings (1)

- En Scala, la cadena de caracteres o String funciona como en Java, por tanto, un string es un objeto immutable.
- Ejemplo:

```
val saludo: String = "Hola, mundo"
```

- Concatenación de Strings.

- `string1.concat(string2);`
- `"Hola," + " mundo" + "!"`

- Longitud de un String

```
var cad = "Hola, mundo!"; var longitud = cad.length();
```

# Strings (2)

- Formateo de un String mediante el método printf()

```
object Prueba {
 def main(args: Array[String]) {
 var floatVar = 12.5
 var intVar = 2000
 var stringVar = "Hola!"
 val fs=printf("El valor de la variable float es "+ "%f\n, el valor de
 la variable entera es %d\n, y el string es" + "is %s",
 floatVar, intVar, stringVar)
 println(fs)
 }
}
```

# Strings (3)

- Interpolación de Strings: El interpolador s

```
object PruebaInterpolacion {
 def main(args: Array[String]) {
 val name = "Pedro"
 println(s"Hola, $name")
 println(s"5 + 3 = ${5 + 3}") }
}
```

# Strings (4)

- Interpolación de Strings: El interpolador f

```
object PruebaInterpolacion {
 def main(args: Array[String]) {
 val peso= 85.5
 val nombre = "Pedro"
 println(f"$nombre%s pesa $peso%2.1f kilos"))
 }
}
```

Output: Pedro pesa 85,5 kilos

# Strings, métodos(1)

java.lang.String

## **char charAt(int indice)**

Devuelve el carácter de la posición indicada

## **int compareTo(Object o)**

Compara un string con respecto de un objeto y devuelve un entero (0> menor, =0 igual, 0< mayor)

## **int compareTo(String anotherString)**

Compara dos string y devuelve un entero (0> menor, =0 igual, 0< mayor)

## **int compareToIgnoreCase(String str)**

Compara dos string ignorando diferencias entre mayúsculas y minúsculas

## **String concat(String str)**

Concatena dos string

## **boolean contentEquals(StringBuffer sb)**

Devuelve verdadero si el string es exactamente igual a la secuencia de caracteres del StringBuffer.

## **String copyValueOf(char[] data)**

Devuelve un string que representa la secuencia de caracteres contenidos en el array.

## **boolean endsWith(String sufijo)**

Comprueba si el string finaliza con el sufijo dado.

## **boolean equals(Object o)**

Función booleana que compara un string con un objeto.

## **boolean equalsIgnoreCase(String anotherString)**

Función booleana que compara un string con un objeto ignorando diferencias entre mayúsculas y minúsculas.

## **void getChars(int ini, int fin, char[] dest, int indice)**

Copia los caracteres de un string comprendidos entre las posiciones ini y fin en el array de caracteres a partir de la posición indice.

## **int hashCode()**

Devuelve un hashcode para el string que invoca el método.

## **int indexOf(int ch)**

Devuelve la posición dentro del string donde se produce la primera ocurrencia del carácter especificado.

# Strings, métodos(2)

java.lang.String

## **int indexOf(int ch, int indice)**

Devuelve la posición dentro del string donde se produce la primera ocurrencia del carácter especificado, comenzando por el índice dado.

## **int indexOf(String subCadena)**

Devuelve el índice dentro del string a partir del cual tiene lugar la primera ocurrencia de la subcadena indicada.

## **int indexOf(String str, int indice)**

Devuelve el índice dentro del string a partir del cual tiene lugar la primera ocurrencia de la cadena pasada como parámetro, comenzando por el índice dado

## **int lastIndexOf(int ch)**

Devuelve la posición dentro del string donde se produce la última ocurrencia del carácter especificado.

## **int lastIndexOf(int ch, int indice)**

Devuelve la posición dentro del string donde se produce la última ocurrencia del carácter especificado, comenzando una búsqueda hacia atrás a partir del índice dado.

## **int lastIndexOf(String subCadena)**

Devuelve la posición dentro del string donde se produce la ocurrencia más a la derecha de la subcadena indicada.

## **int lastIndexOf(String subCadena, int indice)**

Devuelve la posición dentro del string donde se produce la última ocurrencia de la subcadena especificada, comenzando una búsqueda hacia atrás a partir del índice dado.

## **int length()**

Devuelve la longitud del string.

## **boolean matches(String regex)**

Contesta si el string satisface o no una determinada expresión regular.

## **String replace(char oldChar, char newChar)**

Devuelve un nuevo string resultado de reemplazar todas las ocurrencias de oldChar en ese string con newChar.

# Strings, métodos(3)

java.lang.String

UVa

## **String replaceAll(String regex, String subCadena)**

Devuelve un nuevo string resultado de reemplazar todas las coincidencias con la expresión regular en el string mediante la subcadena especificada.

## **String replaceFirst(String regex, String subCadena)**

Devuelve un nuevo string resultado de reemplazar la primera coincidencia con la expresión regular en el string mediante la subcadena especificada

## **boolean startsWith(String prefijo)**

Comprueba si el string comienza con un determinado prefijo.

## **boolean startsWith(String prefix, int indice)**

Comprueba si el string comienza con un prefijo a partir de una determinada posición (índice)

## **String substring(int ini)**

Devuelve un nuevo String que comienza en el índice ini.

## **String substring(int ini, int fin)**

Devuelve un nuevo string que es un substring de este string definido por una posición inicial y otra final-1.

String **slice(int ini, int fin)**. Como substring

## **char[] toCharArray()**

Devuelve un array de caracteres a partir del string

## **String toLowerCase()**

Convierte todos los caracteres del string en minúsculas.

## **String toUpperCase()**

Convierte todos los caracteres del string en mayúsculas.

## **static String valueOf(primitive data type x)**

Devuelve la representación en forma de string del tipo de dato que se pasa como argumento.

## **String trim()**

Devuelve una copia del string eliminando los espacios en blanco de los extremos.

## **String[] split(String T)**

Divide el string en dos piezas y devuelve un array con las dos piezas. T es una expresión regular que representa el símbolo separador. Por ejemplo, separar utilizando el espacio en blanco `S.split("s+")`.

# Tipo Option





# Tipo Option

- El tipo **Option[ T ]** es un contenedor parametrizado para cero o más elementos de un tipo T dado.
- Los valores que puede tomar objeto **Option[T]** puede ser **Some[T]** o **None** (que representa ningún valor devuelto).
- La manera más común de manejar este tipo de objetos es mediante pattern matching:

[illegible]

# Ejemplo con el método `getOrElse`

- **`def getOrElse[B >: A](default: => B): B`**
- Devuelve el valor contenido por el tipo `Option` si la opción es no vacía y el valor por defecto si lo está.

```
object Prueba {
 def main(args: Array[String]){
 val a:Option[Int] = Some(5)
 val b:Option[Int] = None
 println("a contiene: " + a.getOrElse(10))
 println("b contiene: " + b.getOrElse(10)) } }
```

# Ejemplo con el método isEmpty

- **def isEmpty: Boolean**
- Devuelve true si la opción es None y false en caso contrario.

```
object Prueba {
 def main(args: Array[String]) {
 val a:Option[Int] = Some(8)
 val b:Option[Int] = None
 println("a.isEmpty: " + a.isEmpty)
 println("b.isEmpty: " + b.isEmpty) } }
```

# Manejo de Excepciones

# Manejo de Excepciones en Scala

- Como en Java con pequeños matices propios de Scala:

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException
object Prueba {
 def main(args: Array[String]) {
 try {
 val f = new FileReader("texto.txt")
 } catch {
 case ex: FileNotFoundException => { println("fichero no encontrado") }
 case ex: IOException => { println("Excepción I/O") }
 } finally { println("Saliendo del programa") }
 }
}
```