

Scala

Victor Herrero Cazurro

Contenidos

Introduccion	1
Algunas características.....	2
Beneficios.....	2
Documentacion	3
Compilación.....	3
Scala y Java	3
Identificadores	4
Palabras Reservadas	4
Instalacion	5
Consola	5
Primer programa	6
Paquetes	6
Objetos Paquete	7
Imports	8
Variables.....	9
Variables inmutables de solo lectura	9
Ambitos.....	10
Modificadores de Acceso	10
Private	10
Protected	11
Modificacion del ambito de la visibilidad	11
Tipos	12
Tipos Basicos.....	13
Literales	14
Tipo Option	15
String.....	16
Interpolacion	17
Objetos	17
Objeto Compañía	18
Patrón Singleton.....	19
Clase de un Objeto	20
Clases Case (FP).....	20
Constructores Auxiliares	21
Pattern Matching (FP)	21
Sentencias Case condicionales (Guards)	24
Clases.....	24

Constructor Primario	25
Constructores Auxiliares	25
Invocacion del constructor del Padre.....	25
Parametros por defecto	26
Campos	26
Sobreescritura de Get (Accesor) y Set (Mutator)	28
Inicializacion de Campos con Bloques deCodigo	28
Equals y hashCode	29
Clases Internas	30
Objeto Class	31
Sealed Classes	31
Herencia	31
Herencia de campos	32
Sobreescritura	32
Clases Abstractas	33
Campos Abstractos	33
Métodos Abstractos	33
Conversiones (Casting)	34
Traits (FP)	34
Herencia.....	35
Atributos	36
Mixins	36
Funciones	37
Funcion anonima (Lambdas) (FP)	39
Parametros por defecto	40
Paso de parametros por nombre	40
Numero indeterminado de parametros	40
Invocacion Parcial (FP).....	41
Estilo Fluido de programación	41
Funciones de Orden superior (FP).....	42
Retorno de una funcion (FP)	42
Currying (FP)	43
Funcion Predicado.....	44
Closures (FP)	44
Sentencias de Control	45
If-Else	45
For	46
Until	46

To.....	46
Guards	47
For/yield	47
Break/Continue.....	48
Switch	49
Do while	50
While.....	51
Tipos Parametricos (Generics)	51
Covarianza/Contravarianza	52
Colecciones	53
Secuencias (Seq).....	54
IndexedSeq	55
LinearSeq	56
Mapas	57
Sets	58
Procesamiento Paralelo	59
Métodos	59
Métodos de Filtrado	60
Métodos de transformación	62
Operaciones mutables sobre colecciones inmutables	68
Iterators	68
Arrays	68
Arrays Multidimensionales	69
Listas.....	69
Tupla.....	70
Excepciones	70
Try-Catch-Finally	70
Throws	71
I/O.....	71
Creacion de ficheros	71
Lectura de ficheros	71
XML	72
Parseo	72
Busquedas	72
Modificacion	73
Testing.....	74
FunSuite	75
FlatSpec	76

FunSpec	76
WordSpec.....	77
FreeSpec.....	78
Matchers	79
Ejecucion	80
Implicitos	81
Valores Implicitos	81
Métodos Implicitos	82
Clases Implicitas.....	82
Futuros	83
SBT	85
Comandos	85
Key.....	86
Keys predefenidas.....	86
Custom Key	86
Plugins	87
Dependencias	87
Ambitos	88
Actores	88
Respuestas	89

Introduccion

Programacion Imperativa: Paradigma de programacion, que describe la programación en términos del estado del programa y sentencias que cambian dicho estado. La programacion Orienda a Objetos es Imperativa.

Algoritmo para comprobar si en una coleccion con numeros, se tiene el numero 23 escrito en java.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
int numeroABuscar = 23;
boolean tenemosElNumero = false;

for (int numero : numeros) {
    if (numero == numeroABuscar) {
        tenemosElNumero = true;
        break;
    }
}
System.out.println("Tenemos el numero " + numeroABuscar + "? " + tenemosElNumero);
```

Algoritmo equivalente en scala

```
var numeros = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
var numeroABuscar = 23
var tenemosElNumero = false

for (numero <- numeros) {
    println(s"Comprobando el numero $numero")
    if (numero == numeroABuscar) {
        tenemosElNumero = true
        scala.util.control.Breaks.break
    }
}
println(s"Tenemos el numero $numeroABuscar? $esUno")
```

Programacion declarativa: Paradigma de programacion, donde la solución al problema se consigue indicando el problema que se quiere solucionar, no los pasos necesarios para resolverlas. Los lenguajes de programacion declarativa se consideran de mas alto nivel que los imperativos.

Programacion Funcional: Paradigma de programacion declarativa, que se basa en crear funciones matematicas sin estado. Su grandes ventajas son la concurrencia y el paralelismo.

```
System.out.println("Tenemos el color verde?" + colores.contains("verde"));
```

Algunas características

- Lenguaje moderno creado por **Martin Odersky**, influenciado por Java, Ruby, Smalltalk, ML, Haskell, Erlang y otros.
- Orientado a objetos puro, donde toda variable apunta a un objeto (no existen primitivos) y todas las operaciones son métodos.
- Funcional, las funciones son referenciables por variables y se pueden pasar por parametros.
- Corre en la JVM, por lo que se puede emplear todas las librerías que esta incluye.
- No se necesita terminar las sentencias con `;`, solo se empleará cuando haya que separar sentencias en la misma línea, dado que el salto de línea es también identificar de fin de sentencia.
- Se puede prescindir de las llaves `{}` en bloques de código mono-sentencia.
- En la ejecución de funciones se puede prescindir de los `()` si solo se pasa un parametro y el método no está sobrecargado.
- Se puede prescindir del operador `.` para acceder a los miembros de una instancia.
- Es **Case Sensitivity**, los identificadores **Hello** y **hello** son diferentes.
- Por convenio el nombre de las clases debe empezar en mayúsculas y si es compuesto, cada una de las palabras que lo forme, también debe empezar en mayúsculas.
- Por convenio el nombre de los métodos debe empezar en minúsculas y si es compuesto, cada una de las siguientes palabras que lo forme, también debe empezar en mayúsculas.
- No tiene porque existir una **Class** u **Object** definido dentro de un fichero con el mismo nombre del fichero.
- El resultado de la ejecución de un método, será el resultado de la ejecución de la última sentencia, no se emplea **return**.

Beneficios

- Multiparadigma OOP y FP.
- Se ejecuta en la máquina virtual de Java (JVM)
- Legible (siempre que se conozca la sintaxis)
- Conciso (no verboso)

*La diferencia entre emplear Scala o Java con Spark (Big Data) son abrumadoras, el programa en Scala será usualmente 5-10 veces más corto que el programa Java equivalente.

Veamos un ejemplo donde se puede observar la diferencia entre Scala y Java (antes de la versión 8), se trata de un algoritmo que partiendo de un listado de números, obtiene un nuevo listado formado por el subconjunto de elementos que cumplen la condición de ser menores que 4.

```
val nums = List(1,2,3,4,5).filter(_ < 4)

//Donde nums sera un List[Int] = List(1, 2, 3)
```

```
Integer[] intArray = {1,2,3,4,5};
List<Integer> nums = Arrays.asList(intArray);
List<Integer> filteredNums = new LinkedList<Integer>();
for (int n: nums) {
    if (n < 4) filteredNums.add(n);
}

//Donde filteredNums será un LinkedList<Integer> = {1, 2, 3}
```

Documentacion

La referencia del API se puede consultar [aquí](#), así como una amplia bibliografía [aquí](#)

Algunas páginas donde ejecutar código Scala online, como [scalafiddle](#), [scalakata](#) o [scastie](#)

Compilación

Con el SDK, se proporciona el compilador **scalac**, cuyo funcionamiento es similar al compilador de java.

```
scalac HelloWorld.scala
```

Scala y Java

Scala es 100% compatible con el código java, cualquier clase java puede ser referenciada desde Scala.

```
import java.util.Date

object FrenchDate {
    def main(args: Array[String]) {
        val now = new Date
    }
}
```

Todas las clases del paquete **java.lang** son importadas de forma directa.

Se pueden heredar clases e implementar interfaces java en Scala.

NOTE

Eclipse da un error de compilacion al hacer referencia desde codigo Scala a una método estatico de Java, pero en realidad no es problema.

Identificadores

Los identificadores validos, deben empezar por una letra o por `_`, posteriormente pueden aparecer tambien numeros.

No podran empezar por numero, `-` o `$`

Los identificadores validos para los operadores estan compuestos por la sucesion de: `+`, `:`, `?`, `~` o `#`.

```
+  
++  
:::  
<?>  
:>
```

Se pueden crear identificadores compuestos por identificadores alfabeticos y operadores.

```
unary_+  
myvar_ =
```

Tambien se pueden definir identificadores literales, encerrando un **String** entre acentos ```.

```
`x`  
`<clinit>`  
`yield`
```

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	
-	:	=	=>
<-	<:	<%	>:
#	@		

Instalacion

Descargar el SDK de [aquí](#) y el IDE de [aquí](#)

Será necesario tener instalado la jdk 8 de java, que se puede descargar de [aquí](#)

Scala permite trabajar con aplicaciones o con consola, ya que es un lenguaje de scripting.

Consola

Para acceder a la consola, basta con ejecutar el comando **scala.bat**, que se encuentra en a carpeta **bin** de la distribucion de Scala.

Si se desea disponer del comando en cualquier directorio se recomienda definir las siguientes variables de entorno

```
SET SCALA_HOME="D:\utilidades\Scala\scala-2.12.7\"
```

```
SET PATH="%PATH%;%SCALA_HOME%\bin"
```

Para salir de la consola

```
System.exit(-1)
```

Primer programa

De forma analoga a **Java**, en **Scala** el punto de entrada de ejecución es un método **main**, en este caso no de una clase, sino de un **Object (Singleton)**

```
object Programa {  
  def main(args: Array[String]): Unit = {  
  
  }  
}
```

Esto se explica dado que el **main** de Java es estatico, pero en Scala no existen los método estaticos, solo existen los metodos únicos asociados a un **Singleton**.

Otra alternativa a la definición del método **main**, es extender el **object** que representa la aplicación de la clase **App**, definiendo como miembros de la clase las sentencias a ejecutar.

```
object OtroPrograma extends App{  
  println("Hola mundo desde una App!!!!")  
}
```

Paquetes

Es la primera linea no comentada del fichero.

```
package com.ejemplo
```

Se pueden definir empleando bloques de **{ }**

```
package com.ejemplo {  
  
}
```

Con esta sintaxis, se pueden definir varios paquetes en un mismo fichero, o incluso el mismo partido

```
package com.ejemplo.modelo {  
  
}  
package com.ejemplo.logica {  
  
}
```

Se pueden anidar

```
package com {  
    package ejemplo {  
        package modelo {  
        }  
    }  
}
```

Objetos Paquete

Permiten definir métodos, campos y otro código para que este disponible a nivel de paquete, sin tener que definir una **Clase** u **Objeto**.

Para su definición, la convencion, es definir un fichero dentro de un directorio llamado **package.scala**, por ejemplo, si el paquete a definir es **com.ejemplo.modelo**, se ha de crear el fichero **com/ejemplo/modelo/package.scala**.

Dentro del fichero se ha de definir como **package**, el nivel anterior al definido, es decir para el ejemplo seria

```
package com.ejemplo
```

Usando el ultimo nivel, para dar nombre al **Objeto Paquete**

```
package com.ejemplo

package object modelo {

}
```

Dentro se pueden definir los métodos y clases que se quieran, a los que se tendrá acceso directo desde otras **Clases** y **Object** del mismo paquete.

Imports

Los import, permiten en scala al igual que en java, simplificar las referencias a una clase/objeto/trait que esta en otro paquete, dado que las referencias se hacen por el nombre canonico (nombre del paquete + nombre de la clase) y no por el nombre de clase. Los import se añaden en la cabecera de la clase definiendo un alias para las clases dentro del bloque de código.

Así, si se define el siguiente import

```
import java.util.Date
```

Se puede emplear el siguiente código para hacer uso de la clase **java.util.Date**

```
new Date
```

y no

```
new java.util.Date
```

En Scala se pueden crear expresiones con las sentencias de import, pudiendo hacer

- Agrupación de import de clases que están en un mismo paquete

```
import java.util.{Date, Locale}
```

- Importación de todas las clase de un paquete

```
import java.text.DateFormat._
```

- Se pueden renombrar los imports

```
import java.util.{ArrayList => JavaList}

val list = new JavaList[String]
```

Una vez que se renombra el nombre original ya no puede ser empleado, solo sera empleable el alias.

- Exclusión de una clase de una importación masiva

```
import java.util.{Date => _, _}
```

Los imports se pueden poner donde se quieran, siendo locales a su ubicación, pueden estar dentro del cuerpo de una clase u objeto.

```
class Bar {
  def doBar = {
    import scala.util.Random
    println("")
  }
}
```

Variables

Las Variables se definen con la palabra reservada **var**

```
var <nombre de variable> : <tipo> = <valor inicial>
```

El tipo es opcional, ya que se puede inferir del valor, siempre que se asigne un valor, de no asignarse valor, habrá que definir el tipo.

Variables inmutables de solo lectura

Se definen con la palabra reservada **val**.

```
val <nombre de la variable de tipo valor> : <tipo> = <valor>;
```

El tipo es opcional, ya que se puede inferir del valor.

NOTE

Ojo, si se declara una variable de, por ejemplo, tipo Array como **val**, significa que con esa variable no se puede apuntar a otro **Array**, no que no se pueda cambiar el contenido de los elementos del **Array**.

Una variable **val** puede representar una constante, pero para ello deberá de ser de un tipo inmutable.

Ambitos

Existen tres posibles ambitos de las **variables**

- **Campo / Atributo.**

- Su ambito es el objeto al que pertenece, ya sea una instancia de una clase o un Singleton (Object).
- Son accesibles desde cualquier método dentro del objeto y desde fuera de el, siempre que su visibilidad lo permita.
- Puede ser **var** o **val**.

- **Parametros de métodos.**

- Su ambito es el método donde esta definido
- Son inmutables, solo pueden ser **val**.

- **Variables locales.**

- Su ambito es el método donde esta definido
- Puede ser **var** o **val**.

Modificadores de Acceso

El modificador de acceso por defecto es **public**.

Existen a parte **protected** y **private**

Se pueden emplear con **Clases**, **Objetos**, **Miembros** y **Paquetes**.

Private

Solo será accesible desde la clase donde se define.

```

class Outer {
    class Inner {
        private def f() { println("f") }

        class InnerMost {
            f() // OK
        }
    }
    (new Inner).f() // Error: f no es accesible
}

```

Protected

Será accesible desde la clase donde se define y las subclases de esta.

```

package p {
    class Super {
        protected def f() { println("f") }
    }

    class Sub extends Super {
        f()
    }

    class Other {
        (new Super).f() // Error: f no es accesible
    }
}

```

Modificacion del ambito de la visibilidad

Se puede asociar la visibilidad de un miembro a la **Clase**, **Objeto** o **Paquete** que lo contiene, extendiendo o reduciendo el ambito definido.


```

package vida {
  package trabajo {
    class Trabajador {
      private[trabajo] var proyectoAsignado = null
      private[vida] var nombre = null
      private[this] var salario = null

      def hablar(compi : Trabajador) {
        println(compi.proyectoAsignado)
        println(compi.salario) //ERROR: A los compañeros no les cuento mi salario,
que luego tienen envidia ;-)
      }
    }
  }
}

```

Tipos

Todo en Scala es un objeto, incluso las funciones o los numeros, no hay primitivos.

- La expresión

```
1 + 2 * 3 / x
```

- Equivale a

```
(1).+(((2).*(3))./(x))
```

- Se aprecia como +, * o / son funciones de la tipología **Int**, no existe el concepto del operador y se pueden nombrar funciones con caracteres especiales como +, -, etc, que en otros lenguajes estan reservados como operadores, podemos decir que los operadores se pueden sobrescribir.

NOTE

Cuidado con el uso del operador . con los numeros, ya que **1.** se interpreta como **1.0**, será necesaria la sintaxis (1). para acceder a los métodos de **Int**

Como las funciones son objetos, estas pueden ser pasadas por parametro en los métodos

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) {
      callback(); Thread sleep 1000
    }
  }

  def timeFlies() {
    println("time flies like an arrow...")
  }

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

En el ejemplo, el método **oncePerSecond**, recibe como parametro llamado **callback** un método que sin recibir parametros **()**, no retorna nada **Unit**.

Los tipos de los parametros, pueden omitirse en ocasiones, ya que puede inferirse del contexto.

Tipos Basicos

Byte → Entero de 8 bits con signo. Rango desde -128 a 127.

Short → Entero de 16 bits con signo. Rang desde -32768 a 32767.

Int → Entero de 32 bits con signo. Rango desde -2147483648 a 2147483647.

Long → Entero de 64 bits con signo. Rango desde -9223372036854775808 a 9223372036854775807.

Float → Real de 32 bits precision simple.

Double → Real de 64 bits precision doble.

Char → Caracter Unicode de 16 bits sin signo. Range from U+0000 to U+FFFF.

String → Cadena de caracteres.

Boolean → Boleano true/false.

Any → Supertipo de todos los tipos.

AnyVal → Supertipo de todos los tipos Scala cuyo equivalente en Java es primitivo.

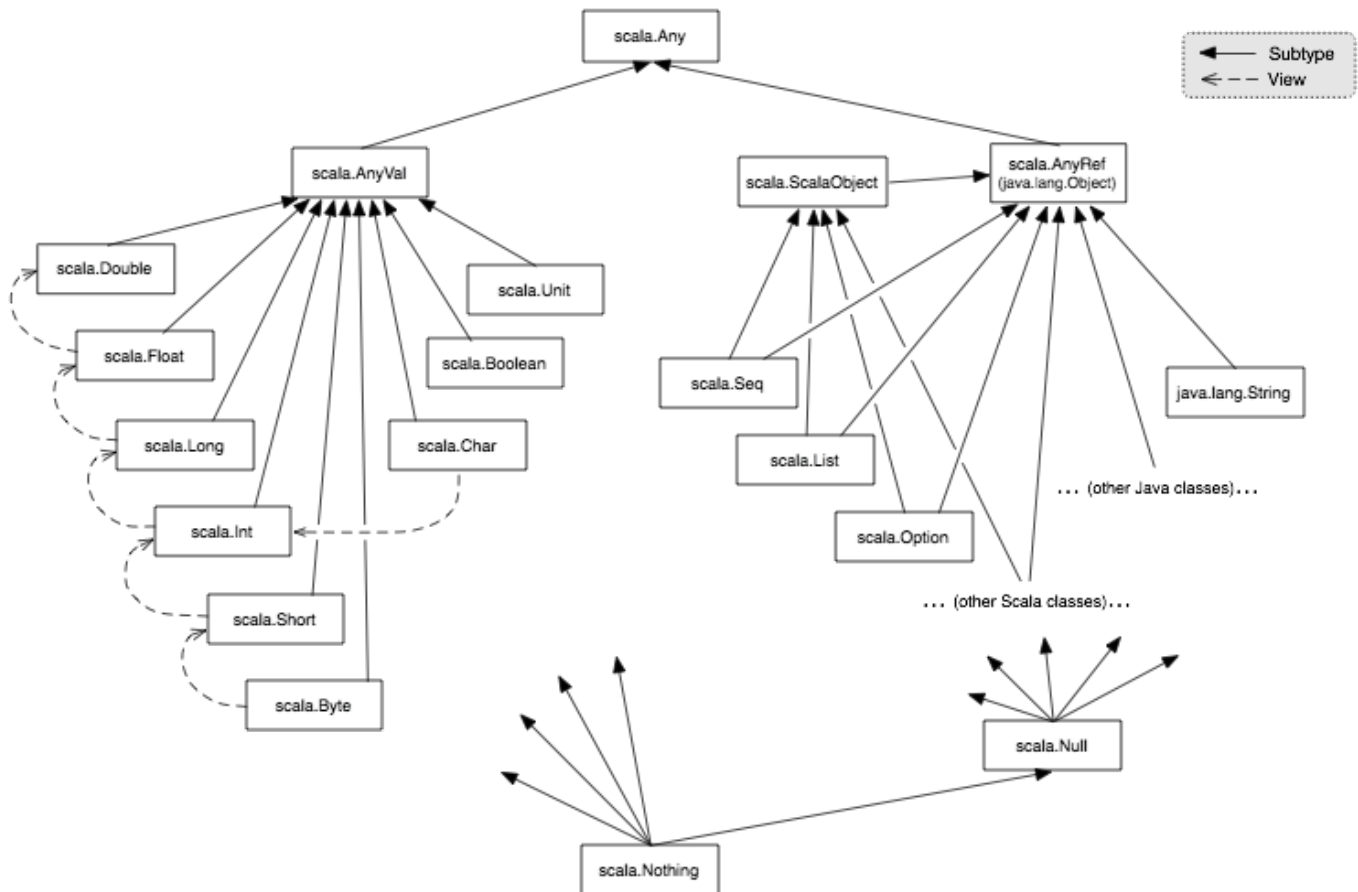
AnyRef → Supertipo de los tipos cuyo equivalente en Java es referencia (equivalente a **Object** en java).

Unit → Sin valor (equivalente a **void** en java).

Null → Trait, de la cual null es una instancia.

Nothing → Trait subtipo de todos los demas, y no es supertipo de nada.

Nil → Lista vacia. Es un **case object** que extiende de List[Nothing]



Literales

- Enteros
 - Son considerados por defecto **Int**, para representar **Long**, se emplean los sufijos **L** o **l**.
 - Aceptan nomenclatura hexadecimal con el prefijo **0x**.
- Reales
 - Son considerados por defecto **Double**, para representar un **Float**, se emplean los sufijos **F** o **f**.
 - Aceptan nomenclatura científica (exponencial) **1.0e100**
- Caracteres
 - Se representan entre comillas simples.
 - Se pueden emplear codigos Unicode.

- Secuencias de escape
 - `\b` - `\u0008` -espacio en blanco
 - `\t` - `\u0009` -tabulacion
 - `\n` - `\u000a` -formfeed FF
 - `\f` - `\u000c` -formfeed FF
 - `\r` - `\u000d` -Retorno de carro
 - `\"` - `\u0022` -comilla doble
 - `'` - `\u0027` -comilla simple
 - `\\` - `\u005c` -contra barra (\)
- Cadenas de caracteres
 - Se representan entre comillas dobles.
 - Se puede definir un **String** en varias lineas empleando la triple comilla doble `"""`

```
"""En un lugar de la mancha,
de cuyo nombre no quiero acordarme
vivía el ilustre hidalgo Don Quijote..."""
```

- `null` → Es un objeto especial de tipo **scala.Null**, para representar que la referencia no está establecida.

Tipo Option

Es un tipo de dato parametrizado, **Option[T]**, que puede tener dos valores

- **None** permite indicar en scala nada.
- **Some(T)** permite encapsular cualquier tipo de objeto para diferenciarlos de **None**

El **Option** permite:

- Declarar un campo sin concretar la inicializacion.

```
case class Persona (var nombre: String, var edad: Int) {
  var genero = None: Option[String]
}
```

- Encapsular el uso de **null**, evitando tener que hacer los procesados de ese valor **null**. Por ejemplo si se define el siguiente objeto

```
val p = new Persona("",0)
```

Se puede acceder al valor de **genero**, como **Option**, estando protegidos de los **null** para lo que se ofrecen los métodos

- **get** → Retorna un **Some(T)** si existe el dato pedido y **None** sino existe.

```
val g = p.genero
```

En este caso el valor de la variable **g** será **None**, si obtuviésemos el valor real, tendríamos un error

```
p.genero.get //Produce un error si genero vale *None*
```

En cambio con el siguiente método nos protegemos del error

- **getOrElse(<default value>)** → Permite obtener un dato si existe, **Some(T)** o el valor por defecto que hemos definido en caso de ser **None**.

```
p.genero.getOrElse("Sin genero definido")
```

Además como los valores de los **Option**, **None** y **Some(T)**, son tipos case, se puede aplicar **Pattern Matching**

```
m.get(1) match {  
  case Some(danceStep) =>  
    println("Soy el amo de la pista bailando: " + danceStep)  
  case None =>  
    println("Soy de los que me quedo mirando en la barra")  
}
```

O de una forma más adaptada a los **Option** con el método **fold**

```
m.get(1).fold(  
  ifEmpty = println("A mi lo de bailar no me va mucho")){  
  step => println(s"Me rompí la cadera bailando $step")  
}
```

String

Interpolacion

El prefijo **s** aplicado a los **String**, permite parametrizar dicho **String** con variables del ambito de Scala.

```
val nombre = "Victor"
println(s"Hola, $nombre")
```

Tambien permite procesar expresiones escritas en Scala

```
val nombre = "Victor"
val apellido = "Herrero"
println(s"Hola ${nombre concat " " concat apellido}")
```

Tambien se proporciona el prefijo **f** que además de lo que hace **s**, permite formatear como se representa la información, las expresiones de formateo, son las misma que en java, se pueden consultar [aquí](#)

```
val nombre = "Victor"
val altura = 1.85d
println(f"$nombre% s mide $altura%2.2f metros")
```

Tambien se proporciona **raw**, que no interpreta las secuencias de escape como **\n** o **\t** ni reglas de formateo.

```
val nombre = "Victor"
val apellido = "Herrero"
val altura = 1.85d
println(f"La persona con nombre:\t$nombre% s y apellidos:\t$apellido% s\nmide $altura%2.2f metros de altura")
println(raw"La persona con nombre:\t$nombre% s y apellidos:\t$apellido% s\nmide $altura%2.2f metros de altura")
```

Los caracteres especiales se han de escapar con ****, salvo el **\$**, que se escapa con **\$\$**

Objetos

En Scala no existe el concepto de **Static**, a cambio existe el concepto de objetos **Singleton**.

Su definición se realiza empleando la palabra reservada **object**

```
object HelloWorld {  
  def main(args: Array[String]) {  
    println("Hello, world!")  
  }  
}
```

No se pueden crear más instancias de los **object**

```
new HelloWorld //No compila
```

Los **object** no reciben parametros de construcción.

```
object HelloWorld (parametro : String){} //No compila
```

Objeto Compañía

Es un **object** en el mismo fichero y con el mismo nombre que una **clase**, que permite asociar a la clase métodos únicos para todas las instancias(**static**)

```
class Persona {  
}  
object Persona {  
}
```

Generalmente se emplea para definir patrones Factoría, haciendo la clase inaccesible con **private** y definiendo métodos **apply** en el objeto de compañía para instanciar la clase.

```
package gente{

  trait IPersona {
    var nombre : String
  }

  private class Persona(var nombre: String) extends IPersona {}

  object Persona {
    println("Contruccion de la Persona")

    def apply(nombre: String): IPersona = {
      new Persona(nombre)
    }
  }
}
```

Pudiendose una vez realizada esta definición, crear objetos Persona con la siguiente sintaxis

```
var p = gente.Persona("Victor")
```

O incluso si se necesita realizar alguna operación para la obtención de los parametros de entrada del método **apply**

```
var p = gente.Persona {
  //Mas sentencias para obtener el String necesario para construir el objeto Persona
  "Victor"
}
```

Patrón Singleton

Para forzar que una clase sea un **Singleton**, lo primero es hacer que el constructor de la clase sea privado, esto en **scala** se consigue añadiendo la palabra reervada **private** entre el nombre de la clase y los parametros

```
class Dios private {

}
```

El siguiente paso sería permitir la ejecucion de un método que permita obtener la única instancia de la clase, pero este método no puede estar asociado a la instancia, en java seria **static** o asociado a la clase, pero en **scala** no existe ese concepto, por tanto se ha de emplear el **companion object**, un **object** con

el mismo nombre de la clase que tiene una relación especial con dicha clase y que si permite definir métodos únicos.

```
class Dios private {  
  
}  
object Dios {  
    val dios = new Dios  
    def getInstance = dios  
}
```

NOTE | Este patrón en **scala**, no tiene sentido implementarlo, ya que **object** ya lo hace.

Clase de un Objeto

Para conseguir el objeto **Class** de un **Objeto**, se dispone del método **getClass**

```
a.getClass
```

Clases Case (FP)

Son una forma que ofrece Scala de definir tipos inmutables.

No emplean la palabra reservada **new** para su construcción, sino el método **apply**, similar al concepto del objeto compañía.

```
case class Persona(nombre: String)  
val p = Person("Juan")
```

Por defecto los atributos miembros definidos son **val** por lo que son solo de lectura y aunque se permite redefinirlos a **var**, esto se desaconseja, ya que para eso están las clases normales.

Se proveen por defecto los siguientes métodos, aunque se pueden definir nuevos, tanto en la clase como en el objeto compañía

- **equals** y **hashCode**: que trabajan sobre la estructura de las instancias y no sobre su identidad
- **toString**: que imprime el valor de una forma **tipo código**
- **apply**: que sustituye al **new**
- **accessors** para todos los atributos. Ya que por defecto los parámetros del constructor son considerados **val**, y en el caso de definirse como **var**, también los **mutator**.

- **unapply**: hace facil el uso de estas clases con **pattern matching**, ya que permite la descomposicion mediante reconocimiento de patrones.
- **copy**: Permite la clonacion de los objetos.

Constructores Auxiliares

No se permiten definir **constructores auxiliares**, dado que realmente cuando se "crea el objeto", hay que recordar que no se usa **new**, no se invoca al constructor, sino que realmente se invoca el método **apply**

```
//Dada la clase
case class Persona (var nombre: String, var edad: Int)

//Es lo mismo
val p = Person("John Smith", 30)

//que
val p = Person.apply("John Smith", 30)
```

Por lo tanto, si se quiere crear otro constructor para una **clase case**, habrá que definir otro método **apply**, como la **clase case** no es redefinible, se emplea un **objeto de compañía** (companion object)

```
case class Persona (var nombre: String, var edad: Int)

//Se emplean los operadores *new* aunque son optativos para distinguir entre la clase y
el objeto singleton
object Persona {
  def apply() = new Persona("", 0)
  def apply(nombre: String) = new Persona(nombre, 0)
}
```

Pattern Matching (FP)

Permiten buscar un patron de coincidencia en un objeto dado, obteniendo resultados distintos dependiendo de lo encontrado.

```
import scala.util.Random

val x: Int = Random.nextInt(10)

x match {
  case 0 => "zero"
  case 1 => "one"
  case 2 => "two"
  case _ => "many"
}
```

Se puede definir un tipo de dato distinto para cada **case**, dado que se contempla que el retorno es de tipo **Any**.

Se puede definir una funcion que implemente la busqueda

```
def matchTest(x: Int): String = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "many"
}

matchTest(3) // many
matchTest(1) // one
```

Se emplean frecuentemente junto con las **Clases Case**

En este ejemplo se busca que el objeto Persona, sea concretamente uno, podriamos decir que se buscan coincidencias a nivel de instancia

```
case class Persona(nombre: String)

var p = Persona("Victor")

p match {
  case Persona("Victor") => println(p.nombre)
  case _ => println("No hay coincidencias")
}
```

En este otro en cambio, nos vale con cualquier objeto de tipo Persona, podríamos decir que se buscan coincidencias a nivel de tipo

```
//Se emplea la propiedad *nombre* del objeto original que se quiere encontrar *p*, que se
emplea para construir una instancia de Persona con la que se compara
p match {
  case Persona(nombre) => println(p.nombre)
  case _ => println("No se encontro el patron")
}
```

Este último caso, es un uso muy habitual junto con las clases case, definiendo con clases case, una jerarquia de herencia, simulando enumerados extensibles.

```
abstract class Notification

case class Email(sender: String, title: String, body: String) extends Notification

case class SMS(caller: String, message: String) extends Notification

case class VoiceRecording(contactName: String, link: String) extends Notification

def showNotification(notification: Notification): String = {
  notification match {

    case Email(email, title, _) => s"You got an email from $email with title: $title"

    case SMS(number, message) => s"You got an SMS from $number! Message: $message"

    case VoiceRecording(name, link) => s"you received a Voice Recording from $name! Click
the link to hear it: $link"
  }
}

val someSms = SMS("12345", "Are you there?")
val someVoiceRecording = VoiceRecording("Tom", "voicerecording.org/id/123")

// prints You got an SMS from 12345! Message: Are you there?
println(showNotification(someSms))

// you received a Voice Recording from Tom! Click the link to hear it:
voicerecording.org/id/123
println(showNotification(someVoiceRecording))
```

Algunos ejemplos de patrones

```

x match {
  // constantes
  case 0 => "zero"
  case true => "true"
  case "hello" => "you said 'hello'"
  case Nil => "an empty List"

  // secuencias
  case List(0, _, _) => "a three-element list with 0 as the first element"
  case List(1, _*) => "a list beginning with 1, having any number of elements"
  case Vector(1, _*) => "a vector starting with 1, having any number of elements"

  // tuplas
  case (a, b) => s"got $a and $b"
  case (a, b, c) => s"got $a, $b, and $c"

  // patrones de construccion
  case Person(first, "Alexander") => s"found an Alexander, first name = $first"
  case Dog("Suka") => "found a dog named Suka"

  // patrones tipados
  case s: String => s"you gave me this string: $s"
  case i: Int => s"thanks for the int: $i"
  case f: Float => s"thanks for the float: $f"
  case a: Array[Int] => s"an array of int: ${a.mkString(",")}"
  case as: Array[String] => s"an array of strings: ${as.mkString(",")}"
}

```

Sentencias Case condicionales (Guards)

Se pueden añadir sentencias **if** a los case de **Pattern Matcher**

```

num match {
  case x if x == 1 => println("one, a lonely number")
  case x if (x == 2 || x == 3) => println(x)
  case _ => println("some other value")
}

```

Clases

Una **clase** en scala, es al igual que en otros lenguajes una plantilla para la creación de objetos.

La definición de **clases** en Scala es muy similar a java, se emplea la palabra reservada **class**.

```
class Persona {  
    //Miembros de la clase  
}
```

Constructor Primario

El constructor primario es la combinacion de los parametros de construcción y las llamadas a métodos y ejecuciones de expresiones y sentencias en el cuerpo de la clase, por tanto al no definirse como tal, no se puede aplicar **sobrecarga** al constructor.

```
class Persona(nombre: String, edad: Int) {  
    println("Construyendo la Persona")  
}
```

Constructores Auxiliares

Se definen como métodos dentro de la clase, con nombre **this**

```
class Persona (var nombre: String, var edad: Int) {  
  
    //Constructor con un parametro  
    def this(edad: Int) {  
        this("", edad)  
    }  
  
    def this(nombre: String) {  
        this(nombre, 0)  
    }  
  
    def this() {  
        this("")  
    }  
}
```

Cada uno de los constructores auxiliares, debe comenzar con la invocación de otro constructor ya sea auxiliar o primario de la misma clase, no se puede invocar desde los auxiliares a constructores del padre.

Invocacion del constructor del Padre

Para la invocación de un constructor del padre, **NO** se emplea la palabra reservada **super**.

Solo se puede invocar al constructor del padre desde el constructor **Primario**, definiendolo en la sentencia de **extends**

```
class Animal (var nombre: String) {  
    // ...  
}  
class Perro (nombre: String) extends Animal (nombre) {  
    // ...  
}
```

Parametros por defecto

Se pueden definir valores por defecto de los parametros del constructor en el constructor primario.

```
class Persona(nombre: String = "Victor", edad: Int = 22) {  
}
```

Con esto se consiguen diversas formas de construcción de los objetos, sin necesidad de definir los constructores auxiliares.

```
new Persona  
new Persona("Juan")  
new Persona("Juan", 35)  
new Persona(edad=35, nombre= "Juan")
```

NOTE

Se puede cambiar el orden de los parametros, siempre que se acompañe del nombre del parametro.

Campos

Existen dos formas de definir campos:

- A partir de los parametros de construcción, acompañándolos de **var** o **val**.

```
class Point(var x: Int, var y: Int) {}
```

- Definiendo una variable en el cuerpo de la clase.

```
class Point{
    var x: Int = 0
    var y: Int = 0
}
```

Para los parametros de construcción no es necesario establecer una asignación, dado que esta vendra dada cuando se construya el objeto, pero para las variables si será necesario.

Se puede hacer un mix, es decir que los parametros de construcción no generen campos, para ello no se les acompaña de **var** o **val** e inicilizar las variables con dichos parametros de construcción.

```
class Point(xc: Int, yc: Int) {
    var x: Int = xc
    var y: Int = yc
}
```

NOTE

Si se definen los campos como **var** seran de lectura/escritura y si se definen como **val** solo de lectura.

Si no se establecen **var** o **val**, es como si se añadiese **private**, los campos son solo accesibles desde el interior de la clase.

```
class Point(private var x: Int, private val y: Int) {
}

//equivale a

class Point(x: Int, y: Int) {
}
```

Al definir los campos con los parametros del constructor, se autogeneran los siguientes métodos, que quedan ocultos y que permiten hacer el get y el set de los campos

```
//Getter o Accesor
def x() : String = {
    return this.x
}

//Setter o Mutator
def x_$eq(x : String) : Unit = {
    this.x = x;
}
```


Sobreescritura de Get (Accesor) y Set (Mutator)

Los métodos aturogenerados no pueden ser sobrescritos, lo que se puede hacer es implementarlos manualmente

```
class Persona(private var _nombre: String = "Victor", private var _edad: Int = 22) {  
    println("Construyendo la Persona: Los Set y Get")  
  
    def nombre = _nombre // accessor  
  
    def nombre_=(nombre: String) { _nombre = nombre } // mutator  
}
```

A resaltar dos situaciones

- El Accesor no define un método cualquiera que no acepta parametros y retorna el atributo privado,

```
def nombre() = {_nombre}
```

sino que tiene una sintaxis especial

NOTE

```
def nombre = _nombre
```

- El mutator, emplea el sufijo =, **en este caso** esta funcionando como caracter especial para representar el espacio en blanco, para poder emplear la sintaxis de asignacion

```
persona.nombre = "Victor" //equivalente a persona.nombre_=("Victor")
```

Este formato obliga a la existencia del accesor para que se interprete el sufijo del mutator, por lo que de no definirlo no se podrá emplear la sintaxis de asignación.

Inicializacion de Campos con Bloques deCodigo

Se puede inicilizar cualquier campo, con el valor generado por la ejecución de un bloque, sabiendo que el valor asignado sera la ultima sentencia del bloque

```
class Persona {
  var nombre = {
    println("Calculando el valor por defecto del campo nombre")
    "Victor"
  }
}
```

Si la inicializacion del campo es muy costosa, se puede definir como **lazy**, pero solo podra afectar a campos **val**

NOTE

```
class Persona {
  lazy val nombre = {
    println("Calculando el valor por defecto del campo nombre")
    "Victor"
  }
}
```

Equals y hashCode

En **Scala** cuando se emplea el operador `==`, se está invocando al método **equals**

```
override def equals(that: Any): Boolean = {true}
```

Se puede aprovechar **Pattern Matching** para implementarlo

```
override def equals(that: Any): Boolean = that match {
  case that: Person => that instanceof [Person] && this.name ==
that.asInstanceOf[Person].name && this.age == that.asInstanceOf[Person].age
  case _            => false
}
override def hashCode: Int = {
  val prime = 31
  var result = 1
  result = prime * result + age;
  result = prime * result + (if (name == null) 0 else name.hashCode)
  return result
}
```

Se dice que toda relacion de igualdad, debe ser

- reflexiva → Se debe cumplir que **x.equals(x)** sea **true**

- simetrica → Se debe cumplir que si **x.equals(y)** son **true**, **y.equals(x)** debe ser **true**
- transitiva → Se debe cumplir que si **x.equals(y)** son **true** y **y.equals(z)** son **true** entonces **z.equals(x)**

Clases Internas

Una clase interna, es aquella que se define dentro del ambito de otra.

```
class OuterClass {
    class InnerClass {
        var x = 1
    }
}
```

En **Scala** las **Clases** internas estan rodeadas por **Objetos** externos, esto quiere decir que al instanciar un nuevo objeto de una clase interna, se ha de hacer partiendo de un objetos de la clase contenedora. Si se hace a traves de una variable, esta tiene que ser **val**.

```
val oc1 = new OuterClass
val ic1 = new oc1.InnerClass
```

Se pueden definir clases internas de **Singleton**

```
object OuterObject {
    class InnerClass {
        var x = 1
    }
}
```

Siendo la instanciacion mas directa, ya que no es necesario instanciar el **Singleton**

```
val ic1 = new OuterObject.InnerClass
```

Tambien se pueden definir **Singleton** dentro de las clases

```
class OuterClass {
    object InnerObject {
        val y = 2
    }
}
```

Teniendo que crear una instancia de la clase contenedora para acceder al **Singleton**

```
var io = new OuterClass().InnerObject
```

Objeto Class

Para conseguir el objeto **Class** de una clase, se dispone del método **classOf**

```
classOf[TargetDataLine]
```

Sealed Classes

Los **Traits** y las **classes** pueden ser marcadas con la palabra reservada **sealed** lo cual indica que todos los subtipos de dicha clase, deben estar en el mismo fichero, lo cual hace que sean conocidos por el desarrollador de la clase **sealed**.

```
sealed abstract class Furniture
case class Couch() extends Furniture
case class Chair() extends Furniture

def findPlaceToSit(piece: Furniture): String = piece match {
  case a: Couch => "Lie on the couch"
  case b: Chair => "Sit on the chair"
}
```

Herencia

Se puede heredar de otras clases de forma analoga a java, con la palabra reservada **extends**

```
class Persona {
  override def toString = "Texto que representa en formato string el objeto"
}

class Cliente extends Persona {
}
```

Todas las clases en Scala heredan de otra, en caso de no definirse, será de **scala.AnyRef**

```
class Trabajo extends AnyRef {  
  
}
```

Herencia de campos

Al definir en el constructor parametros con **var** o **val**, se crean campos y al heredar de una clase con campos, estos parametros se han de proporcionar, la sintaxis para ellos seria

```
class Persona(var nombre: String) {  
  
}  
  
class Cliente (nombre: String, var edad: Int) extends Persona (nombre) {  
  
}
```

Se puede comprobar como cada clase tiene sus campos, en este caso nombre es un campo mantenido por la clase **Persona**, por lo que en los parametros de la clase **Cliente** no se le indica **var**, de hacerlo daria error de compilación, porque se intentan crear el **accesor** y el **mutator** y ya estan definidos en el padre.

Sobreescritura

La sintaxis para la sobreescritura se basa en la palabra reservada **override**

```
override def toString = "" + re + (if (im < 0) "" else "+") + im + "i"
```

Se pueden sobrescribir tanto métodos como campos definidos en la clase padre

```
abstract class Animal {  
    val sonido = "No disponible" // provide an initial value  
    def hablar { println(sonido) }  
}  
class Perro extends Animal {  
    override val sonido = "Guau" // override the value  
}
```

Para los metodos y los campos abstractos, no es necesario poner **override**

Clases Abstractas

Dado que en **Scala** existen los **Traits** que son mas potentes que las clases abstractas, estas quedan relegadas a

- Cuando quieras crear una clase con parametros de construcción.
- El codigo tenga que ser invocado desde Java.

Se definen con la palabra reservada **abstract**

```
abstract class Persona (name: String) {  
  
}
```

Campos Abstractos

Las clases abstractas pueden tener campos no inicializados

```
abstract class Persona (nombre: String) {  
    var edad : Int  
}
```

Cuando en las clases hijas se implementen estos campos, se debera volver a indicar **val** o **var**, ya que el campo no se crea en la clase abstracta, solo se crean el **accesor** y el **mutator**

```
class Trabajador (nombre: String) extends Persona(nombre) {  
    var edad = 12  
}
```

Métodos Abstractos

Las definiciones de métodos abstractos que no aceptan parametros, se puede reconvertir en un campo en la definición de la clase hija.

```
abstract class Mascota (nombre: String) {  
    def sonido: String  
}  
class Perro (nombre: String) extends Mascota (nombre) {  
    val sonido = "Guau"  
}
```

Conversiones (Casting)

Se proporciona el método **asInstanceOf[T]** que permite realizar la conversion de tipos

```
var p = new Persona  
  
var p1 = p.asInstanceOf[Persona]
```

De no pertenecer a la jerarquia la clase a la que se quiere convertir, dará un **ClassCastException**.

Por supuesto, se puede aplicar tambien para los tipos numericos, dado que no existen los primitivos

```
val a = 10 //a: Int = 10  
  
val b = a.asInstanceOf[Long] //b: Long = 10  
  
val c = a.asInstanceOf[Byte] //c: Byte = 10
```

Todos los tipos numericos tienen métodos para convertir el objeto actual a otro tipo numerico, incluido **char**: `toByte`, `toChar`, `toDouble`, `toFloat`, `toInt`, `toLong` y `toShort`

Traits (FP)

La palabra **trait** en inglés puede traducirse literalmente como rasgo o característica.

Son similares a las interfaces de Java, salvo porque los **Traits** pueden ser parcialmente implementados, pueden definir implementaciones por defecto para algunos métodos y pueden heredar de clases.

Se emplea la palabra reservada **trait** para definirlos.

Si los métodos definidos por el **Trait** no tienen parametros, se puede obviar la estructura de los parentesis

```
trait BaseSoundPlayer {  
  def play  
  def close  
  def pause  
  def stop  
  def resume  
}
```

En contraste con las clases, los **Traits** no pueden tener parámetros de constructor.

```
trait Similarity {  
  def isSimilar(x: Any): Boolean  
  def isNotSimilar(x: Any): Boolean = !isSimilar(x)  
}
```

Herencia

Un **trait** puede heredar de otro **trait** o de una **clase**, para lo cual se emplea la palabra reservada **extends**

```
class StarfleetComponent  
trait StarfleetWarpCore extends StarfleetComponent
```

En el caso de heredar de una clase, el **Trait** solo podrá ser empleado en aquellas clases que de heredar, lo hagan de la misma que el **Trait**.

```
class StarfleetComponent  
trait StarfleetWarpCore extends StarfleetComponent  
class RomulanStuff  
  
// Esta herencia no compila  
class Warbird extends RomulanStuff with StarfleetWarpCore
```

Para implementar un **Trait** se emplea la palabra reservada **extends**, si solo se implementa un **trait**

```
class Point(xc: Int, yc: Int) extends Similarity {  
  var x: Int = xc  
  var y: Int = yc  
  
  def isSimilar(obj: Any) =  
    obj.isInstanceOf[Point] &&  
    obj.asInstanceOf[Point].x == x  
}
```

En cambio si la clase extiende de otra y además implementa algún **trait**, se emplea la palabra reservada **with**

```
class Foo extends BaseClass with Trait1 with Trait2 {}
```


NOTE

Salvo que la clase que implemente el **Trait** sea abstracta, debe implementar todos los metodos que el **trait** no implemente.

Atributos

Los **traits** pueden definir atributos, que pueden a su vez ser abstractos o concretos

```
trait PizzaTrait {  
    var numToppings: Int // abstracto  
    var size = 14 // concreto  
    val maxNumToppings = 10 // constante concreta  
}
```

Se consideran concretos cuando son inicializados y abstractos cuando no.

```
class Pizza extends PizzaTrait {  
    var numToppings = 0 // no se necesita 'override'  
    size = 16 // no se necesita ni 'var' ni 'override'  
    override val maxNumToppings = 10 // Se necesita 'override' y 'val' por ser una  
    constante  
}
```

Las clases que extiendan los **traits** deberan inicializar los atributos no inicializados o deberan ser abstractas.

Mixins

La palabra mixin puede ser traducida como mezcla.

Permite componer una clase con **Métodos** definidos en distintos **lugares** (1 clase y n traits)

Se emplea la palabra reservada **with** para indicar los **Traits** de los que extenderá la clase.

```

trait Tail {
    def wagTail { println("tail is wagging") }
    def stopTail { println("tail is stopped") }
}

abstract class Pet (var name: String) {
    def speak // abstract
    def ownerIsHome { println("excited") }
    def jumpForJoy { println("jumping for joy") }
}

class Dog (name: String) extends Pet (name) with Tail {
    def speak { println("woof") }
    override def ownerIsHome {
        //Se emplean los métodos del Trait
        wagTail
        speak
    }
}

```

Funciones

Las Funciones se definen con la palabra reservada **def**, siguiendo la sintaxis

```

def <nombre funcion>(<variable>: <tipo>, ...) : <Tipo retornado> = {
    <cuerpo de la funcion>
}

```

No es obligatorio definir un tipo de retorno, se considera el valor de la última sentencia ejecutada, siempre que no sea de asignación.

```

def retornaUnString(saludo : String) = {saludo}

```

No es obligatorio influir las llaves, si la función es mono-sentencia

```

def retornaUnInt(edad: Int) = edad

```

No es obligatorio incluir los parentesis (), sino hay argumentos

```
def retornaUnString = {"Hola!"}
```

De no incluir los parentesis, la invocación de la función, debera ser tambien sin parentesis

NOTE

```
retornaUnString //Invocacion valida  
retornaUnString() //Invocacion invalida
```

No es obligatorio incluir el =, si el tipo retornado es **Unit**

```
def noRetornaNada {"Hola!"}
```

No es obligatorio es uso del operador . para acceder a miembros de variables, por tanto la expresion

```
xs filter (pivote >)
```

Es equivalente a

```
xs.filter(pivote >)
```

Se pueden omitir los parentesis en la invocación de una función, siempre que

- se referencie al objeto dueño de la función.
- que sea la última invocación de la sentencia.
- dicho método acepte un unico parametro.

```
Funciones retornaUnInt 12
```

```
Funciones retornaUnString "Juan"
```

Las funciones se pueden anidar unas dentro de otras, pudiendo accederse los parametros y variables de la contenedora, desde la contenida

```
def sort(xs: Array[Int]) = {
    def swap(i: Int, j: Int) {
        val t = xs(i); xs(i) = xs(j); xs(j) = t
    }
}
```

Funcion anonima (Lambdas) (FP)

Se pueden definir funciones anonimas como la del siguiente ejemplo

```
(i: Int) => {i % 2 == 0}
```

En este caso es una función que acepta un **Int** y retorna un **Booleano**

De formas mas reducida, se puede escribir

```
_ % 2 == 0
```

Dado que se puede emplear `_` para referenciar de forma generica al unico parametro recibido por la función.

Aplicando un funcion anonima en un trozo de programa.

```
object EjemploFuncionAnonima {
    def funcionDeOrdenMayor(funcionPasadaPorParametros : (String, String) => String) {
        println(funcionPasadaPorParametros("Juan", "Martinez"))
    }

    def main(args: Array[String]) {
        funcionDeOrdenMayor((nombre : String, apellido : String) => {nombre + apellido} )
    }
}
```

Donde se emplea la siguiente sintaxis en la definición de una nueva funcion anonima

```
(<parametros separados por ,>) => {<cuerpo de la función>}
```

En el ejemplo la funcion anonima definida será

```
(nombre : String, apellido : String) => {nombre + apellido}
```

Parametros por defecto

Se pueden definir parametros por defecto para los métodos

```
def saludo (nombre: String = "Mundo"){  
    print("Hola " + nombre + "!!!!")  
}
```

Por lo que se puede invocar un método sin cumplir con los parametros que se piden

```
saludo("Victor")  
  
saludo()
```

Paso de parametros por nombre

Se pueden pasar los parámetros en cualquier orden, empleando los nombres de los parametros para su asignacion.

```
object Saludador {  
    def depura(prefijo: String = "Hola", sufijo: String = "!!!!", nombre: String) {  
        println(prefijo + nombre + sufijo)  
    }  
    def main(args: Array[String]) {  
        depura(sufijo = "?????", nombre = "Victor")  
    }  
}
```

Numero indeterminado de parametros

Se puede definir un método para que acepte un numero indeerminado de parametros con *

```
def printAll(strings: String*) {  
    strings.foreach(println)  
}
```

Todas las siguientes invocaciones serán validas

```
printAll()
printAll("foo")
printAll("foo", "bar")
printAll("foo", "bar", "baz")
```

Como en java y otros lenguajes, solo puede existir un parametro indeterminado en numero y de haber mas parametros, el indeterminado será el último.

Si los elementos se quieren sacar de un Array, List, Seq, Vector, ... se puede indicar con `_*` la transformación para que sea valida la siguiente sentencia

```
val fruits = List("apple", "banana", "cherry")

printAll(fruits: _*)
```

Invocacion Parcial (FP)

En Scala se puede preparar la invocación de un método indicando solo algunos de sus parametros, pudiendo realizar la invocación de dicho método mas adelante, ya si, indicando los parametros restantes.

Esto se consigue con una variable intermedia que guarda la preparación y el comodin `_`

```
val punteroASumar2 = sumar(2, _: Int)
```

Donde la definicion del método sería

```
def multiplica (a: Int, b: Int) : Int = {
    return a * b
}
```

Y la invocación real del algoritmo

```
punteroASumar2(1)
```

Estilo Fluido de programación

Cuando se precisa retornar el mismo objeto que fue invocado para permitir la concatenación de métodos, se puede emplear **this.type**

```
val person = new Person.setFirstName("Al").setLastName("Alexander")

class Person {
    protected var fname = ""
    protected var lname = ""

    def setFirstName(firstName: String): this.type = {
        fname = firstName
        this
    }
    def setLastName(lastName: String): this.type = {
        lname = lastName
        this
    }
}
```

Funciones de Orden superior (FP)

Son funciones que reciben por paramtro otras funciones

```
def exec(f: (String) => Unit, name: String) {
    f(name)
}
```

En este caso se recibe por parametro una función **f** que recibe un **String** y no retorna nada.

Normalmente si la funcion recibida recibe parametros, estos tambien llegan como parametros.

Es uno de los ejes de la **Programación Funcional**

Esta definicion da paso a las **Closures**.

Retorno de una funcion (FP)

Se puede retornar como resultado de la ejecucion de un método, una funcion, un algoritmo.

```
def saySomething(prefix: String) = (s: String) => {
    prefix + " " + s
}
```

En este caso, se esta retornando una funcion tal que

```
(s: String) => { prefix + " " + s }
```

Por lo que al ejecutar

```
val sayHello = saySomething("Hello")
```

Se tiene la referencia a la funcion anonima, aunque en este caso ya no es anonima, la cual a su vez es invocable.

```
sayHello("Al")
```

Currying (FP)

Este comportamiento deriva en el **currying**, patrón que permite definir una nueva función que adapta el uso de otra funcion existente, es decir dada un funcion que presenta una interface (datos de entrada y datos de salida), la adapta a los requerimientos representados por otra funcion con una nueva interface(datos de entrada y datos de salida).

En el ejemplo anterior, se esta adaptando un algoritmo que presenta dos variables **prefix** y **s** a un formato, en el que se puede predefinir el **prefix**, para reutilizarlo con varios **s**

Se puede emplear esta funcionalidad asociada con los **Matcher Pattern** para crear algo similar a una factoria

```
def greeting(language: String) = (name: String) => {  
  language match {  
    case "english" => "Hello, " + name  
    case "spanish" => "Buenos dias, " + name  
  }  
}
```

La clave de ese método, reside en la firma = **(name: String)** ⇒ {}, la aparicion de este codigo en la firma indica que se retornará una función que acepta un **String**

Se puede ver mas claro


```
def greeting(language: String) = (name: String) => {
  val english = () => "Hello, " + name
  val spanish = () => "Buenos días, " + name

  language match {
    case "english" => println("returning 'english' function")
                      english()
    case "spanish" => println("returning 'spanish' function")
                     spanish()
  }
}
```

Funcion Predicado

Las funciones predicado son funciones que reciben uno o mas parametros y retornan un **booleano** definen una expresión que recibe parametros y genera un valor de un tipo.

```
//def isEven (i: Int) = if (i % 2 == 0) true else false
def isEven (i: Int) = (i % 2 == 0)
```

Closures (FP)

Es una función que puede interaccionar no solo con las variables de su ambito de ejecución, sino con las de su ambito de definición.

```

object AplicacionClosures extends App {

    //Se define una variable
    var hello = "Hello"

    //Se define un metodo, que usa la anterior variable, que representa la closure, ya que
    //tendra acceso cuando se ejecute a
    //la variable *hello*, asi como al parametro *name* que se le pase en ejecución
    def sayHello(name: String) { println(s"$hello, $name") }

    // Se crea un objeto con un método que acepta un Closure
    val foo = new otherscope.Foo

    //Se ejecuta el método, pasando el metodo definido anteriormente como Closure
    foo.exec(sayHello, "Al")

    //Se cambia la variable
    hello = "Hola"

    //Se ejecuta de nuevo el metodo con la misma closure
    foo.exec(sayHello, "Lorenzo")
}

package otherscope {
    class Foo {
        def exec(f: (String) => Unit, name: String) {
            f(name)
        }
    }
}

```

En el ejemplo **f** es la **closure**, en su ambito tiene la variable **name**, pero cuando se ejecuta el codigo subyacente, tambien se hace uso de otra variable **hello** que esta en otro ambito, que puede ser modificada en ese otro ambito y cambiar el resultado de la ejecución de la **closure**.

Sentencias de Control

Como todas las operaciones en Scala son **Métodos**.

If-Else

Similar a java, pero en **Scala** dentro de la estructura se considera la devolucion de un resultado, como el operador ternario de java.

```
val x = if (a) y else z
```

For

Método que permite aplicar una funcionalidad a todos los elementos de una Secuencia.

```
for (i <- Array(1,2,3)) println(i)
```

En el ejemplo la función a aplicar es **println(i)** y la secuencia es **Array(1,2,3)**, donde la variable **i** apunta a cada elemento de la secuencia en cada iteración.

Se pueden emplear las palabras reservadas **until** y **to** para establecer rangos de ejecución.

- **until**: No incluye el último elemento
- **to**: incluye el último elemento

Until

Por ejemplo, **until** se suele emplear para recorrer colecciones, dado que si empieza en 0 y se da como extremo el tamaño, no hace falta indicar el clásico **a.length - 1**

```
var a = Array(1,2,3)
for (i <- 0 until a.length) {
    println(s"$i is ${a(i)}")
}
```

To

Y **to** para definir los extremos del rango

```
for (i <- 1 to 3) {
    println(i)
}
```

Esta sentencia, realmente se traduce en

```
1.to(3).foreach(((i) => println(i)))
```

Se pueden definir varios contadores, produciéndose la ejecución del bucle con todas las combinaciones posibles de los contadores

```
for (i <- 1 to 2; j <- 1 to 2) println(s"i = $i, j = $j")
```

Aunque la sintaxis mas habitual es

```
for {  
  i <- 1 to 3  
  j <- 1 to 5  
  k <- 1 to 10  
} println(s"i = $i, j = $j, k = $k")
```

Guards

Tambien se pueden establecer condiciones en los rangos (**guards**) con la sentencia **if**

```
for (i <- 1 to 10 if i < 4) {  
  println(i)  
}
```

Esta sentencia, realmente se traduce en

```
1.to(3).withFilter(((i) => i.<(4))).foreach(((i) => println(i)))
```

NOTE | Se pueden concatenar varios **if**

Otra sintaxis habitual seria

```
for {  
  i <- 1 to 10  
  if i < 4  
} println(i)
```

For/yield

Método que retorna una Secuencia del mismo tipo que la que se está procesando con el **for**, pero a la que aplica a cada elemento la transformación definida por **yield**

```
val arrayresultado = for (i <- Array(1,2,3)) yield i * 2  
  
//Donde arrayresultado será un Array[Int] = Array(2, 4, 6)
```

En el ejemplo la secuencia es **Array(1,2,3)** y la transformación aplicada a cada elemento de la secuencia es **i x 2**

Internamente se ejecuta el método **map()** de la secuencia.

NOTE

```
for {  
  i <- 1 to 10  
} yield i  
  
//Equivale a  
  
1.to(10).map(((i) => i))
```

Si se necesita un algoritmo mas complejo que una unica sentencia, la sintaxis seria

```
val lengths = for (e <- names) yield {  
  e.length  
}
```

Donde la última sentencia es el valor incluido en la nueva coleccion.

Break/Continue

No existen las palabras reservadas **break** y **continue**, a cambio ofrece una clase **scala.util.control.Breaks**

```
scala.util.control.Breaks.breakable(  
  for (i <- 0 to array.length - 1) {  
    if (i > 1) {  
      scala.util.control.Breaks.break // corta la ejecucion del bucle  
    }  
    println(i + array(i))  
  }  
)
```

Se podria escribir de la siguiente forma si se incluye **import scala.util.control.Breaks._**

```
breakable(
  for (i <- 0 to array.length - 1) {
    if (i > 1) {
      break // corta la ejecucion del bucle
    }
    println(i + array(i))
  }
)
```

Para expresar un **continue**, se emplea los mismos métodos, pero sin que afecten al bucle

```
for (i <- 0 to array.length - 1) {
  breakable {
    if (array(i).contains("verde")) {
      println("Se ha encontrado el color Verde!!!!")
      break //termina el if, pero sigue con el for
    }
    println("Un color mas -> " + array(i))
  }
}
```

Si se desea emplear etiquetas asociadas a los **break** se han de utilizar instancias la clase **scala.util.control.Break**

```
val Inner = new Breaks
val Outer = new Breaks
Outer.breakable {
  for (i <- 1 to 5) {
    Inner.breakable {
      for (j <- 'a' to 'e') {
        if (i == 1 && j == 'c') Inner.break else println(s"i: $i, j: $j")
        if (i == 2 && j == 'b') Outer.break
      }
    }
  }
}
```

Switch

No existe la sentencia como tal, sino que se emplean los **pattern matcher**

```

val month = i match {
  case 1 => "January"
  case 2 => "February"
  case 3 => "March"
  case 4 => "April"
  case 5 => "May"
  case 6 => "June"
  case 7 => "July"
  case 8 => "August"
  case 9 => "September"
  case 10 => "October"
  case 11 => "November"
  case 12 => "December"
  case _ => "Invalid month" // the default, catch-all
}

```

El caracter `_` se emplea como **comodin** para el resto de valores, si se desea procesar el valor con el que se ha comparado en el **case** en las sentencias de la derecha, se puede asociar el **case** con una variable

```

val month = i match {
  case 1 => "January"
  case 2 => "February"
  case 3 => "March"
  case 4 => "April"
  case 5 => "May"
  case 6 => "June"
  case 7 => "July"
  case 8 => "August"
  case 9 => "September"
  case 10 => "October"
  case 11 => "November"
  case 12 => "December"
  case invalid => invalid + " is a Invalid month" // the default, catch-all
}

```

Do while

La sentencia no difiere de java

```
var a = 10;

do {
  println("Value of a: " + a);
  a = a + 1;
} while (a < 20)
```

While

La sentencia no difiere de java

```
var a = 10;

while( a < 20 ){
  println( "Value of a: " + a );
  a = a + 1;
}
```

Tipos Parametricos (Generics)

Permiten definir clases que emplean variables de tipos sin especificar.

```
class Stack[T] {
  var elems: List[T] = Nil
  def push(x: T) { elems = x :: elems }
  def top: T = elems.head
  def pop() { elems = elems.tail }
}
```

Estas variables de los tipos, deben ser especificadas en el momento de la instanciación

```
val stack = new Stack[Int]
```

Se pueden aplicar reglas sobre los tipos que afecten a su jerarquia, así se puede indicar que

- Un tipo debe pertenecer a una jerarquia con el operador <:


```

class Animal {}
class Perro extends Animal {}

//El <: indica que T debe ser un subtipo de Animal
class Lista[T <: Animal] {}

var listaPerro = new Lista[Perro]

var listaAnimal = new Lista[Animal]

//No esta permitido por ejemplo
//var listaAny = new Lista[Any]

```

- Un tipo debe ser supertipo de otro con el operador >:

```

class Animal {}

class Lista[T >: Animal] {}

var listaAny = new Lista[Any]

var listaAnimal = new Lista[Animal]

//No esta permitido por ejemplo
//var listaPerro = new Lista[Perro]

```

Covarianza/Contravarianza

Además el API permite definir varianza y contravarianza aplicada a los tipos, de tal forma que se cumpla que

- **Covarianza:** Si un tipo A (Perro), es subtipo de otro B (Animal), otro tipo parametrizado con A (List[Perro]), será subtipo de ese mismo tipo parametrizado con B (List[Animal]), pudiendo por tanto referenciar con variables del tipo parametrizado con B (List[Animal]) a objetos del tipo parametrizado con A (List[Perro]).

Para definir la covarianza se emplea el prefijo +

```

class Animal {}
class Perro extends Animal {}

class Lista[+T] {}

var listaPerro = new Lista[Perro]

var listaAnimal = new Lista[Animal]

listaAnimal = listaPerro

```

- **Contravarianza** → Si un tipo A (Animal), es subtipo de otro B (Any), otro tipo parametrizado con B (List[Any]), será subtipo de ese mismo tipo parametrizado con A (List[Animal]), pudiendo por tanto referenciar con variables del tipo parametrizado con A (List[Animal]) a objetos del tipo parametrizado con B (List[Any]).

Para definir la contravarianza se emplea el prefijo -

```

class Animal {}

class Lista[-T] {}

var listaAny = new Lista[Any]

var listaAnimal = new Lista[Animal]

listaAnimal = listaAny

```

Colecciones

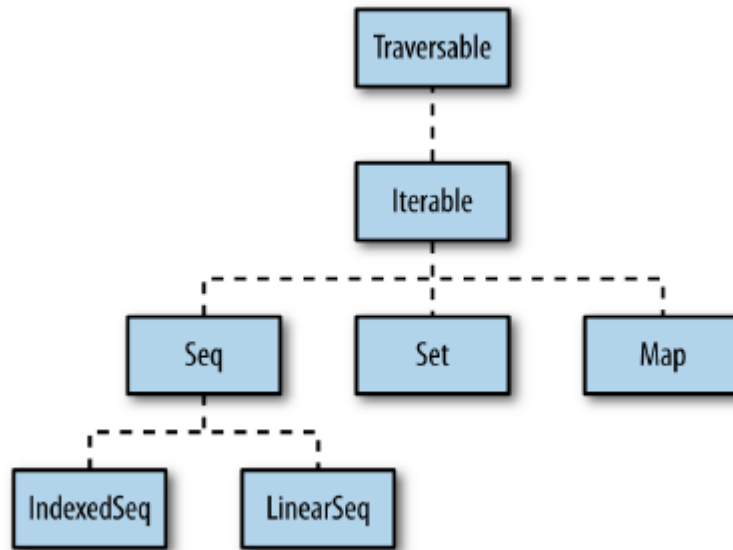
Los conceptos mas importantes a la hora de trabajar con **Colecciones** en **Scala** son

- Las hay **mutables** e **inmutables**, las mas habituales de emplear en FP son las inmutables de las cuales se proporcionan alias y por tanto no se necesita añadir **imports**.
- Predicado
- Funcion anonima
- Bucles implicitos

Las colecciones son parametrizables en cuanto al tipo de datos que contienen

```
List[Number](1, 2.0, 33D, 400L)
// List[java.lang.Number] = List(1, 2.0, 33.0, 400)
```

El API es extenso, una vision general, no lleva a los principales **Traits**

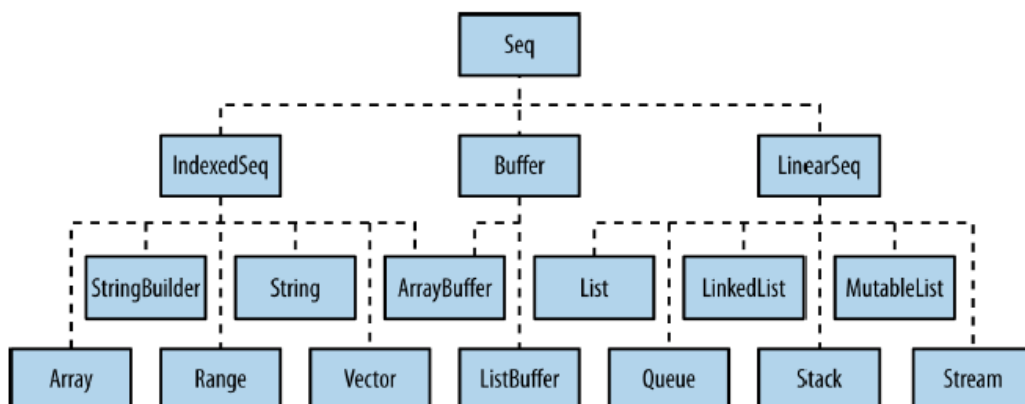


Como se ve el elemento de mayor jerarquia es **Transaversable**, siendo la implementacion por defecto **List**

```
val fruits = Traversable("apple", "banana", "orange")
// fruits = Traversable[String] = List(apple, banana, orange)
```

Secuencias (Seq)

Empezemos por las secuencias



Este API se divide principalmente en dos secciones

- **IndexedSeq** → Colecciones que permiten el acceso aleatorio a los elemento de forma eficiente, por ejemplo los **Array**. Por defecto emplea **Vector** como implementacion

```
val x = IndexedSeq(1,2,3)

//x = Vector(1, 2, 3)
```

- **LinearSeq** → Colecciones que son eficientes a la hora de dividir en **head** (primer elemento) y **tail** (resto). Por defecto implementa **List** que es una lista enlaza.

```
val seq = scala.collection.immutable.LinearSeq(1,2,3)
// seq = List(1, 2, 3)
```

IndexedSeq

Colecciones que permiten el acceso aleatorio a los elemento de forma eficiente, por ejemplo los **Array**, por defecto emplea **Vector** como implementacion.

Las implementaciones de referencia son

- Vector → Inmutable

```
val a = Vector(1, 2, 3)
// a = Vector(1, 2, 3)
val c = a.updated(0, 7)
// a = Vector(1, 2, 3)
// c = Vector(7, 2, 3)
```

- ArrayBuffer → Mutable

```
var nums = ArrayBuffer(1, 2, 3)
//nums = scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 3)
nums += 4
//nums = scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 3, 4)
```

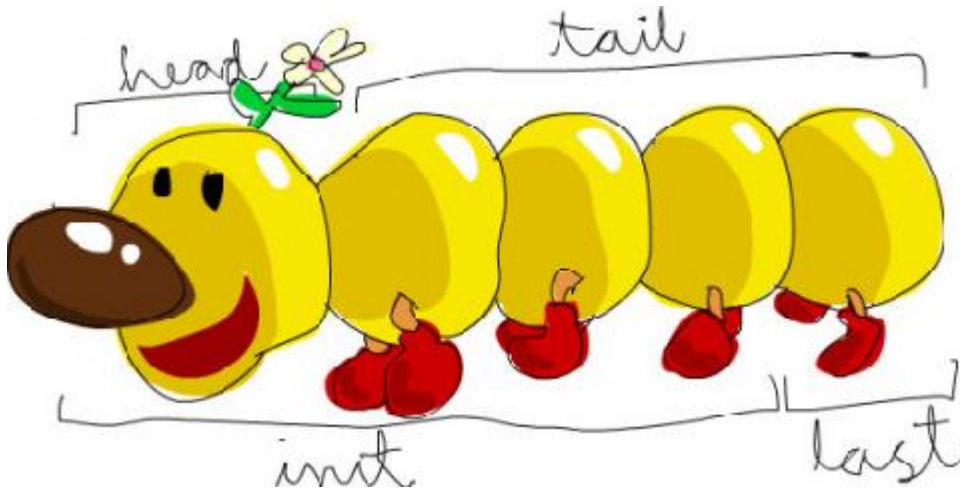
Además de **Vector** y **ArrayBuffer**, el API ofrece otras implementaciones como

- Range → Rango de valores enteros (inmutable)
- String → secuencia de caracteres indexada (inmutable)
- Array → Array de Java, que puede cambiar sus elementos, pero no de tamaño (mutable)

- ArrayStack → Estructura LIFO que mejora el rendimiento de **Stack** (mutable)
- StringBuilder → Permite construir **Strings** (mutable)

LinearSeq

Colecciones que son eficientes a la hora de dividir en **head** y **tail** o en **init** y **last**, por ello están pensadas para la recursividad.



Por defecto implementa **List** que es una lista enlazada.

Las implementaciones de referencia son

- List → Inmutable
- ListBuffer → Mutable

Además de **List** y **ListBuffer**, el API ofrece otras implementaciones como

- Queue → Estructura de datos FIFO (mutable / inmutable)
- Stack → Estructura de datos LIFO (mutable / inmutable)
- Stream → Similar a **List**, pero es perezosa y persistente. Su método **tail** es **lazy**. Es ideal para secuencias muy largas. (inmutable)

```
scala> val stream = (1 to 100000000).toStream
//Se conoce el primer elemento, pero el resto no, es lazy
//stream: scala.collection.immutable.Stream[Int] = Stream(1, ?)

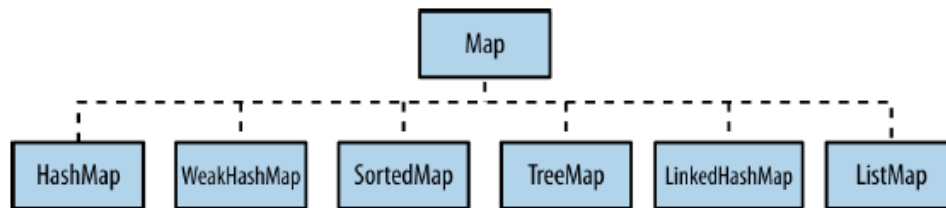
//Si se obtiene el primero, es inmediato
scala> stream.head
res0: Int = 1

//Pero si se obtiene el resto, solo se obtiene un nuevo stream con el segundo elemento
del original que ahora es el primero de este nuevo stream
scala> stream.tail
res1: scala.collection.immutable.Stream[Int] = Stream(2, ?)
```

- LinkedList → Como **List** pero mutable.
- DoubleLinkedList → Como **LinkedList** pero bidireccional (mutable)

Mapas

Colecciones **clave/valor** donde las **claves** son unicas.



Tambien los hay mutables e inmutables

```
val m = Map(1 -> "a", 2 -> "b")

//scala.collection.immutable.Map(1 -> a, 2 -> b)

val m = collection.mutable.Map(1 -> "a", 2 -> "b")

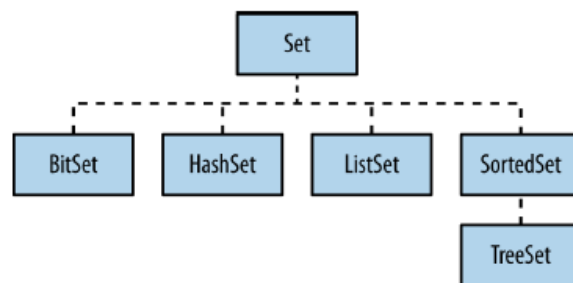
//scala.collection.mutable.Map(2 -> b, 1 -> a)
```

El API ofrece las siguientes implementaciones

- `HashMap` → Emplea **HashTree** y **HashTable** para su implementación (mutable / immutable)
- `LinkedHashMap` → Retorna los elementos en orden de inserción (mutable)
- `ListMap` → Internamente emplea un `LinearSeq`. Los elementos son insertados en la cabecera de la colección, por lo que los retorna en orden inverso a la inserción (mutable / immutable)
- `SortedMap` → Igual que **TreeMap**
- `TreeMap` → Implementación de árbol rojo-negro (immutable)
- `WeakHashMap` → Implementa **WeakHashMap** de Java, que no guarda referencias directas a los objetos que se le pasan como claves, cuando las claves no se referencian desde fuera, los elementos del **WeakHashMap** se pierden.

Sets

Colecciones que no admiten repetidos.



```

val set = Set(1, 2, 3)

//scala.collection.immutable.Set(1, 2, 3)

val s = collection.mutable.Set(1, 2, 3)

//scala.collection.mutable.Set(1, 2, 3)
  
```

El API ofrece las siguientes implementaciones

- `BitSet` → Set de enteros positivos. Permite optimizar el uso de la memoria para los Set de enteros. (mutable / immutable)
- `HashSet` → Emplea **HashTree** y **HashTable** para su implementación (mutable / immutable)
- `LinkedHashSet` → Retorna los elementos en orden de inserción (mutable)
- `ListSet` → implementado con un `List` (immutable)

- TreeSet → Emplea un SortedSet
- SortedSet →

Procesamiento Paralelo

El API proporciona las siguientes implementaciones de colecciones que procesan los datos de forma paralela

- ParHashMap
- ParHashSet
- ParIterable
- ParMap
- ParRange
- ParSeq
- ParSet
- ParVector

El procesamiento en paralelo, divide los elementos de la coleccion para el procesamiento, al concluir todos los procesos, los recombina para obtener la solucion.

No se recomienda su uso para casos en los que el orden del procesamiento de los elementos es importante

```
val v = Vector.range(0, 10)
v.foreach(print)
//0123456789
v.par.foreach(print)
//0231564789
```

Y si se recomienda, en aquellos en los que el orden no es importante, sino el resultado final

```
val v = Vector.range(0, 10)
v.sum
//45
v.par.sum
//45
```

Métodos

Para trabajar con las colecciones, se dispone de un gran abanico de métodos, que se pueden dividir en

grupos segun la funcionalidad que proporcionan

Métodos de Filtrado

- `collect f` → Aplica **pattern matching** a los elementos de la coleccion, generando una nueva coleccion con los resultados.

```
scala> List("hello", 1, true, "world") collect { case s: String => s }  
// List[String] = List(hello, world)
```

- `diff c` → Retorna la difefencia entre la coleccion desde la que se invoca y la coleccion que se pasa por parametro.

```
val low = (1 to 5) toSet  
  
val medium = (3 to 7) toSet  
  
val min = low.diff(medium)  
// Set(1, 2)
```

- `distinct` → Retorna una coleccion en la que se han quitado los duplicados

```
List(1,2,3,2,1).distinct  
// List(1, 2, 3)
```

- `drop n` → Retorna un nueva coleccion donde se ha eliminado los primeros n elementos, donde n es un numero que se pasa por parametro.

```
val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
numbers.drop(5)  
// List[Int] = List(6, 7, 8, 9, 10)
```

- `dropWhile p` → Retorna una coleccion en la que se ha borrado el primer elemento que satisface el predicado.

```
val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
numbers.dropWhile(_ % 2 != 0)  
// List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10)
```

- `filter p` → Método que retorna una Secuencia del mismo tipo que la que se procesa, pero que aplicando a cada elemento la condicion definida, incluye en la Secuencia retornada solo aquellos

elementos de la Secuencia inicial que cumplan la condicion.

Firma del método

```
def filter(p: T => Boolean): Array[T]
```

Ejemplo de uso

Ejemplo que emplea _ para referenciar a cada item de la Secuencia.

```
val resultado = List(1,2,3,4,5).filter(_ < 4)

//La anterior expresion equivale a
//val resultado = List(1,2,3,4,5).filter(i => i < 4)

//Donde resultado será un List[Int] = List(1, 2, 3)
```

```
val numeros = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

val resultado = numeros.filter(_ % 2 == 0)

//La anterior expresion equivale a
//numeros.filter((i: Int) => i % 2 == 0)

//Donde resultado será List[Int] = List(2, 4)
```

Un ejemplo de uso con mapas

```
val extensions = Map("steve" -> 100, "bob" -> 101, "joe" -> 201)

val filtradas = extensions.filter((namePhone: (String, Int)) => namePhone._2 < 200)

// Donde filtradas será scala.collection.immutable.Map[String,Int] = Map((steve,100),
(bob,101))

//Empleando patter matching

val filtradas = extensions.filter({case (name, extension) => extension < 200})

// Donde filtradas será scala.collection.immutable.Map[String,Int] = Map((steve,100),
(bob,101))
```

- filterNot p → Retorna nacoleccion con todos los elementos que no cumplen el predicado.

- `find p` → Retorna un **Option** (`Some[T]`) del primer elemento que encuentra que cumpla el predicado, y **None** si no lo encuentra.
- `foldLeft(n)(op)` → Aplica una operacion a los elementos, comenzando por uno concreto y recorriendo la coleccion de izquierda a derecha
- `foldRight(n)(op)` → Aplica una operacion a los elementos, comenzando por uno concreto y recorriendo la coleccion de derecha a izquierda
- `head` → Retorna el primer elemento de la coleccion o **NoSuchElementException** si esta vacia.
- `headOption` → Retorna el primer elemento de la coleccion como un `Option (Some[T])` o **None** si esta vacia.
- `init` → Retorna los primeros elementos de la coleccion, exceptuando el ultimo. Retorna un **UnsupportedOperationException** si esta vacia.
- `intersect c` → Retorna una coleccion con los elementos comunes a dos colecciones.
- `last` → Retorna el ultimo elemento de la coleccion o **NoSuchElementException** si esta vacia
- `lastOption` → Retorna el ultimo elemento de la coleccion como un `Option (Some[T])` o **None** si esta vacia.
- `reduceLeft op` → Aplica una operacion a los elementos comenzando por el primero
- `reduceRight op` → Aplica una operacion a los elementos, comenzando por el ultimo
- `slice (from, to)` → Retorna el trozo de coleccion desde (from) hasta (to)
- `tail` → Retorna todos los elementos de la coleccion excepto el primero
- `take n` → Retorna los primeros n elementos de la coleccion
- `takeWhile p` → Retorna elementos mientras el predicado se cumpla parando en el momento que no se cumpla.
- `union c` → Retorna la coleccion resultante de unir dos colecciones.

Métodos de transformación

- `+= x` → Añade un elemento a la coleccion
- `+= (x,y,z)` → Añade todos los elementos a la coleccion
- `++= c` → Añade todos los elementos de la coleccion a la coleccion inicial
- `--= x` → Borra un elemento de la coleccion
- `--= (x,y,z)` → Borra todos los elementos de la coleccion
- `--= c` → Borra todos los elementos de la coleccion en la coleccion inicial
- `c(n) = x` → Asigna un valor al elemento x de la coleccion
- `++ c` → Crea una nueva coleccion con los elementos de ambas colecciones.
- `:+ e` → Crea una nueva coleccion en la que ha añadido el elemento e

- `+: e` → Crea una nueva coleccion en la que ha añadido el elemento `e` al comienzo
- `e :: list` → Retorna una List a la que al cominezo ha añadido `e` (solo con List)
- `append(e1, e2)` → Añade al final de la coleccion los elementos indicados
- `appendAll c` → Añade al final de la coleccion los elementos de la nueva coleccion
- `insert(n, e1, e2)` → Inserta los elementos a partir de la posicion `n`
- `insertAll(n, c)` → Añade los elementos de la nueva coleccion a la coleccion existente a partir de la posicion `n`
- `prepend(e1, e2)` → Añade al principio de la coleccion los elementos indicados
- `prependAll c` → Añade al principio de la coleccion los elementos de la nueva coleccion
- `updated(e1, e2)` → Retorna una nueva coleccion con los elementos indicados añadidos
- `clear` → Borra todos los elementos de la coleccion
- `remove n` → Borra el elemento en la posicion `n`
- `remove(n, len)` → Borra el numero de elementos indicado con `len` desde la posición `n`
- `flatMap` → Aplica a cada elemento de la coleccion un funcion (como `map`), retornado la coleccion aplanada de la transformacion realizada. Se emplea sobre colecciones que contienen colecciones, para transformar y aplanar a la vez.

```
val nestedNumbers = List(List(1, 2), List(3, 4))
// nestedNumbers: List[List[Int]] = List(List(1, 2), List(3, 4))

nestedNumbers.flatMap(x => x.map(_ * 2))
//res0: List[Int] = List(2, 4, 6, 8)
```

- `map f` → Método que retorna una Secuencia del mismo tipo que la que se procesa, pero a la que aplica a cada elemento la transformación definida, en este caso se emplea el caracter reservado `_` para referenciar a cada elemento de la Secuencia.

```
val arrayResultado = Array(1,2,3).map(_ * 2)

//Donde arrayResultado será un Array[Int] = Array(2, 4, 6)
```

- `reverse` → Retorna la coleccion en orden inverso al actual.
- `sort` → Retorna una coleccion estableciendo el orden por defecto, de menor a mayor

```
val xs = Seq(1, 5, 3, 4, 6, 2)

xs.sorted //1,2,3,4,5,6
```

- `sortWith` → Retorna una colección con los elementos ordenados según la función definida, esta función, tendrá dos parámetros de entrada, el elemento actual y el elemento siguiente

```
val xs = Seq(1, 5, 3, 4, 6, 2)

xs.sortWith(_<_) //1,2,3,4,5,6
```

- `sortBy f` → Ordena por un campo descrito por la función, válido para objetos complejos

```
case class Person(val name:String, val age:Int)

val ps = Seq(Person("John", 32), Person("Bruce", 24), Person("Cindy", 33),
Person("Sandra", 18))

ps.sortBy(_.age)
```

- `zip c` → Crea una colección formada por **Pair** o **Tuple2** compuestos con los elementos de las dos colecciones

```
List(1, 2, 3).zip(List("a", "b", "c"))
// List[(Int, String)] = List((1,a), (2,b), (3,c))
```

- `unzip` → Crea dos colecciones partiendo de una colección de elementos de tipo **Tuple2**

```
List((1,"a"), (3, "b"), (4, "d")).unzip
// (List[Int], List[String]) = (List(1, 3, 4),List(a, b, d))
```

- `zipWithIndex` → Como **zip**, pero trabaja con el índice de la colección, creando las tuplas como (elemento, posición)

```
List("a", "b", "c").zipWithIndex
// List[(Int, String)] = List((a,0), (b,1), (c,2))
```

Se puede emplear junto con el bucle `for`

```
val numeros = List.range(1, 10)

for ((item, posicion) <- numeros.zipWithIndex) {
  println(s"$posicion is $item")
}
```

- Métodos de agrupación:
 - `groupBy p` → Retorna un mapa con dos colecciones, donde una colección contiene los elementos que no lo cumplen el predicado y la otra los que si lo cumplen.
 - `partition p` → Retorna dos colecciones acorde al predicado, una que cumple y otra que no cumple.
 - `sliding`
 - `span p` → Retorna una colección de dos colecciones, la primera es la que cumple con **takeWhile p**, y la segunda la que cumple con **dropWhile p**
 - `splitAt n` → Retorna una colección con dos colecciones, que son el resultado de partir por el elemento n.
- Métodos informativos:
 - `canEqual`
 - `contains`
 - `containsSlice`
 - `count` → cuenta todos los elementos que cumplen una determinada función predicado
 - `endsWith`
 - `exists` → retorna True si el predicado se cumple para algún elemento
 - `forall p` → retorna True si el predicado se cumple para todos los elementos
 - `hasDefiniteSize` → Retorna true si la colección tiene un tamaño definido, retorna false para los **Stream**
 - `indexOf` →
 - `indexOfSlice`
 - `indexWhere`
 - `isDefinedAt`
 - `isEmpty` → Retorna true si la colección está vacía.
 - `lastIndexOf`
 - `lastIndexOfSlice`
 - `lastIndexWhere`
 - `max` → Retorna el mayor elemento de la colección
 - `min` → Retorna el menor elemento de la colección
 - `nonEmpty` → Retorna true si la colección no está vacía
 - `product` → Retorna la multiplicación de todos los elementos de la colección
 - `segmentLength`
 - `size` → retorna el tamaño de la colección

- `startsWith`
- `sum` → Retorna la suma de todos los elementos de la coleccion
- Otros:
 - `par` → Crea una coleccion de procesamineto paralelo de la coleccion.
 - `view` → Crea una coleccion perezosa de la coleccion. Ideal para procesamiento de grandes cantidades de datos

```
(1 to 1000000000).view.filter(_ % 2 == 0).take(10).toList
//List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

- `flatten` → Convierte una lista de listas, en un unica lista.

```
List(List(1, 2), List(3, 4)).flatten
// List[Int] = List(1, 2, 3, 4)
```

- `foreach f` → Itera sobre los elementos de la coleccion, aplicando una funcion a cada uno

```
val x = Vector(1, 2, 3)

//Se define una lambda con mas de una sentencia
x.foreach (item => {
  val a = item
  println(a.toHexString)
})

//Se define una lambda monosentencia
x.foreach(i => println(i))

//Se puede ahorrar la declaracion del parametro que le llega al a expresion lambda
x.foreach(println(_))

//De ser una unica funcion, No hace falta indicar que parametro se pasa, se pasa
automaticamente el item
x.foreach(println)
```

- `mkString` → Crea un String de la coleccion.
- `range(n, m)` → Permite popular una coleccion con un rango de valores numerico, indicando un paso

```
Array.range(1, 5)
// Array[Int] = Array(1, 2, 3, 4)
Vector.range(0, 10, 2)
// collection.immutable.Vector[Int] = Vector(0, 2, 4, 6, 8)
```

- Solo para Maps

- - k → Retorna un nuevo mapa, donde se ha eliminado el elemento con clave k
- - (k1, k2, k3) → Retorna un nuevo mapa, donde se han eliminado los elementos con claves k1, k2 y k3
- — c → Retorna un nuevo mapa, donde se han borrado los elementos cuyas claves estan definidas en la coleccion
- += (k → v) → Añade un nuevo elemento al mapa
- += (k1 → v1, k2 → v2) → Añade dos nuevos elementos al mapa
- ++= c → Añade los elementos de la coleccion al mapa
- ++= List(3 → "c")
- -= k → Borra el elemento del mapa con clave k
- -= (k1, k2, k3) → Borra los elementos del mapa con claves k1, k2 y k3
- -= c → Borra los elementos del mapa tomando como claves los elementos de la coleccion
- m(k) → Retorna el valor asociado a la clave k
- get k → Retorna el valor asociado a la clave K como Option (Some[T])
- isDefinedAt k → Retorna tru si el mapa contiene la clave k
- keys → Retorna las claves como Iterable
- KeySet → Retorna la claves con Set
- mapValues f → Retorna un nuevo map aplicando f a todos los valores
- values → Retorna os valores del map como Iterable.

Es facil con un bucle **for** iterar sobre un **Map**, dado que lo unico que hay que indicar es que las variables que apuntan a cada elemento, son una **Tupla**.

```
val names = Map("fname" -> "Robert", "lname" -> "Goren")
for ((k,v) <- names) {
  println(s"key: $k, value: $v")
}
```


Operaciones mutables sobre colecciones inmutables

Se puede producir la siguiente situación, añadir elementos sobre una colección inmutable, porque se referencia con **var**, en vez de con **val**

```
var sisters = Vector("Melinda")
// sisters: collection.immutable.Vector[String] = Vector(Melinda)
sisters = sisters :+ "Melissa"
//sisters: collection.immutable.Vector[String] = Vector(Melinda, Melissa)
sisters = sisters :+ "Marisa"
//sisters: collection.immutable.Vector[String] = Vector(Melinda, Melissa, Marisa)
```

No es que la colección sea inmutable, sino que se están creando nuevas colecciones cada vez que se añade un elemento, es decir lo anterior es similar a

```
var sisters = Vector("Melinda")
sisters = Vector("Melinda", "Melissa")
sisters = Vector("Melinda", "Melissa", "Marisa")
```

Lo que no se podría realizar es

```
var sisters = Vector("Melinda")
sisters(0) = "Molly"
```

Iterators

Son un tipo de colección, no se emplean como en Java con un bucle while y el método hasNext, sino que tienen métodos como el resto.

```
val it = Iterator(1,2,3)
it.foreach(println)
```

Una de los grandes usos que se le da, es la transformación a otro tipo de colección con los métodos: toArray, toBuffer, toIndexedSeq, toIterable, toIterator, toList, toMap, toSeq, toSet, toStream, toString y toTraversable.

Arrays

Los arrays se definen con la sintaxis **Array[T]** en lugar de **T[]**

```
def sort(xs: Array[Int]) {}
```

Los accesos a los elementos del array, se escriben **a(i)** en lugar de **a[i]**.

```
xs(0);
```

Los Arrays, son instancias de tipo **Seq[T]**, por lo que se dispone de todas las funciones de esta tipología.

Arrays Multidimensionales

Se definen con el método **ofDim**

```
val array = Array.ofDim[Int](2,2)
array(0)(0) = 0
array(0)(1) = 1
array(1)(0) = 2
array(1)(1) = 3
```

Se pueden recorrer con un bucle for y tantos contadores como dimensiones.

```
for {
  i <- 0 to 1
  j <- 0 to 1
} println(s"($i)($j) = ${array(i)(j)}")
```

Listas

Son un tipo de coleccion especial, que esta compuesto por celdas contiguas y que finalizan con el valor **Nil**

Se pueden definir de dos formas, la primera con **List**

```
val x = List(1, 2, 3)
```

La segunda mas primitiva

```
val y = 1 :: 2 :: 3 :: Nil
```

Tupla

Tipo de dato que permite manejar objetos de distinta tipología.

Permite por ejemplo manejar mas de un objeto sin tener que definir una tipología particular, muy útil a la hora de retornar datos en un método.

```
def sumaRestaMultiplicacion(x: Int, y: Int): (Int, Int, Int) = {  
    return (x + y, x - y, x * y)  
}
```

Se pueden definir unicamente con **(,)** o empleando **Tuple** seguido del numero de elementos de la Tupla

```
Tuple3(1,2,3)
```

Se han definido hasta **Tuple22**

Para acceder a los elementos de la tupla se emplea la sintaxis **_num**

```
val t = (3, "Three", new Person("Al"))  
// t = (Int, java.lang.String, Person) = (3,Three,Person(Al))  
t._1  
// Int = 3
```

Excepciones

En Scala todas las excepciones son **unchecked** es decir no necesitan ser procesadas por código.

Try-Catch-Finally

Similar a java, pero en **Scala** se puede emplear **pattern matching** para las clausulas **Catch**

```
val s = "Foo"  
try {  
    val i = s.toInt  
} catch {  
    case e: Exception => e.printStackTrace  
    case e: FileNotFoundException => println("Couldn't find that file.")  
    case e: IOException => println("Had an IOException trying to read that file")  
}
```

Throws

Para indicar al objeto invocante que una función lanza excepciones, se puede hacer es anotar con **@throws** los métodos para avisar a sus invocantes de que lanza excepciones

```
@throws(classOf[Exception])
def method {

}
```

I/O

Para trabajar con ficheros en Scala, se emplea en API de java, junto con alguna clase de apoyo de Scala.

Creacion de ficheros

*Algoritmo que crea un fichero **test.txt** en la raíz del proyecto, con contenido **Hello Scala***

```
import java.io._

object Ficheros extends App {
    val writer = new PrintWriter(new File("test.txt" ))

    writer.write("Hello Scala")
    writer.close()
}
```

Lectura de ficheros

Algoritmo para leer línea a línea un fichero, convirtiendo los valores leídos en enteros y multiplicándolos por 2

```
import scala.io.Source

object Ficheros extends App {
    Source.fromFile("test.txt").getLines().map(_.toInt * 2).foreach {
        println
    }
}
```

XML

Scala puede trabajar con XML de forma nativa a través del objeto **scala.xml.Elem**, pudiendo escribir código como el siguiente

```
val x = <b>Hello World!</b>
```

La representación del XML se realiza con las clases **Seq[Node]**, **Node** y **NodeSeq**.

En la definición de un XML, se puede referenciar a variables, invocar métodos o definir expresiones

```
def now = System.currentTimeMillis.toString
val xml = <b time={now}>Hello World</b>
val other = <b>{if (true) "dogs" else ""}</b>
```

Por defecto se escapan los caracteres especiales

```
<b attr={"Hello < Zoo"}/>

res6: scala.xml.Elem = <b attr="Hello &lt; Zoo"></b>
```

Se puede evitar el escapado del contenido de los XML con **PCData()**

```
<script>{PCData(info)}</script>
```

Parseo

Con el uso del paquete **scala.xml** se pueden cargar y parsear xml remotos

```
import scala.xml._
val xml = XML.load("http://demo.liftweb.net/")
```

Busquedas

Se pueden realizar búsquedas de nodos empleando los operadores

- **** (double backslash): Que encuentra todas las etiquetas en el documento con el nombre indicado. Retornando un **NodeSeq**

Busca todas las etiquetas de tipo <a>

```
xml \\ "a"
```

- \ (single backslash): Que encuentra todas las etiquetas con el nombre indicado que directamente sean hijas.

Busca todas las etiquetas de tipo <a> hijas directas del nodo principal

```
xml \ "a"
```

- "@<texto>*: Permite buscar un atributo de la etiqueta.

Busca todos los atributos href de las etiquetas hijas directas del nodo principal

```
xml \ "@href"
```

Dado que **Seq[Node]** es una coleccion se pueden aplicar operaciones mas avanzadas para las busquedas

Selecciona todas las etiquetas <a>, cada etiqueta la transforma en su atributo href, cada atributo lo transforma en su valor textual y de ellos se queda con aquellos cuyo valor textual empieza por http:

```
(xml \\ "a").map(_ \ "@href").map(_.text).filter(_ startsWith "http:")
```

Otra opcion de conseguir lo anterior.

```
(xml \\ "a").filter(n => (n \ "@href").text.startsWith("http:"))
```

Modificacion

Se proporcionan diversas clases que permiten realizar modificaciones sobre un xml, las cuales se encuentran en el paquete **scala.xml.transform**

Algunas de ellas son

- RuleTransformer: Permite aplicar una regla de transformacion a un XML.
- RewriteRule: Clase base para definir comportamientos de modificacion de XML.

```
import scala.xml._
import scala.xml.transform._
val removeIt = new RewriteRule {
  override def transform(n: Node): NodeSeq = n match {
    case e: Elem if (e \ "@instruction").text == "remove" => NodeSeq.Empty
    case n => n
  }
}

val xmlBooks =
  <books instruction="update">
    <book instruction="remove" name="book1" status=""/>
    <book instruction="add" name="book2" status=""/>
  </books>

new RuleTransformer(removeIt).transform(xmlBooks)
```

Siendo el resultado del anterior código, la eliminación del nodo con **instruction="remove"**

```
res0: Seq[scala.xml.Node] =
<books instruction="update">
<book instruction="add" status="" name="book2"></book>
</books>
```

También se puede añadir contenido al XML

```
val addIt = new RewriteRule {
  override def transform(n: Node): NodeSeq = n match {
    case e: Elem if (e \ "@instruction").text == "add" =>
      new Elem(e.prefix, e.label, e.attributes.remove("instruction"), e.scope,
transform(e.child) ++ <added>I added this</added> :_*)
    case n => n
  }
}
```

Testing

En la Programación funcional, el testing de las funcionalidades, ofrece mayor garantía que en la programación imperativa, dado que en la funcional el resultado de la ejecución de una función, solo dependen de los parámetros de entrada, y no del ámbito de ejecución.

El Framework para la realización de test en **Scala** se llama **scalatest**, se puede encontrar su documentación [aquí](#)

Se necesitará añadir al classpath el jar de **scalatest**

Para añadirlo con **SBT** se ha de añadir al fichero **/build.sbt**

```
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.1" % "test"
```

Para añadir con **Maven** la librería para el caso en el que se emplee la versión 2.11 de Scala, sería

```
<dependency>
  <groupId>org.scalatest</groupId>
  <artifactId>scalatest_2.11</artifactId>
  <version>3.0.1</version>
  <scope>test</scope>
</dependency>
```

Existen varios tipos de Estilos de definir los Test

- **FunSuite**: Define funciones `test(<enunciado>)`, ejecutándose todas.
- **FlatSpec**: fusiona la definición del test y del enunciado, instrumentalizando el enunciado con palabras reservadas como **should**, **can**, **must**, etc y referenciando al **SUT** con **it** cuando se definen varios test sobre el mismo **SUT**.
- **FunSpec**: Define el **SUT** con funciones **describe()**, aplicando los enunciados con funciones **it()**
- **WordSpec**: Permite anidar la definición de enunciados instrumentalizados con **when**, **should**, **can**, **must**, etc.
- **FreeSpec**: Permite partir los enunciados textuales para su reutilización.
- **PropSpec**
- **FeatureSpec**

NOTE	Ojo!!! con realizar pruebas sobre funciones definidas dentro de un object que extienda de App, que presenta un comportamiento no esperado, no inicializando los atributos del object.
-------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

FunSuite

Se define una clase que extienda **FunSuite**, donde se definirá la lógica del **Test** y se incluirán tantos **test** como sean necesarios


```
import org.scalatest.FunSuite

class SetSuite extends FunSuite {

  test("An empty Set should have size 0") {
    assert(Set.empty.size == 0)
  }

  test("Invoking head on an empty Set should produce NoSuchElementException") {
    assertThrows[NoSuchElementException] {
      Set.empty.head
    }
  }
}
```

Los **test** definen un texto descriptivo del test, que ha de facilitar la comprensión de la prueba que implementan y un cuerpo con las aserciones necesarias.

FlatSpec

Se define una clase que extienda **FlatSpec**, definiendose una serie de enunciados donde se emplean **should**, **can**, **must**, ...

```
import org.scalatest.FlatSpec

class SetSpec extends FlatSpec {

  "An empty Set" should "have size 0" in {
    assert(Set.empty.size == 0)
  }

  it should "produce NoSuchElementException when head is invoked" in {
    assertThrows[NoSuchElementException] {
      Set.empty.head
    }
  }
}
```

Si se necesitan mas de un test sobre el mismo elemento, se puede emplear **it**, en el ejemplo anterior **it** referencia a **An empty Set**.

FunSpec

Se define una clase que extienda **FunSpec**, definiendose una serie de enunciados empleando la

anidacion de **describe**, para la reutilizacion de dichos enunciados, definiendo la asercion dentro de un **it**

Familiar para desarrolladores de **Ruby RSpec**

```
import org.scalatest.FunSpec

class SetSpec extends FunSpec {

  describe("A Set") {
    describe("when empty") {
      it("should have size 0") {
        assert(Set.empty.size == 0)
      }

      it("should produce NoSuchElementException when head is invoked") {
        assertThrows[NoSuchElementException] {
          Set.empty.head
        }
      }
    }
  }
}
```

WordSpec

Se define una clase que extienda **WordSpec**, definiendose una serie de enunciados empleando los verbos: **should**, **must**, o **can** asociados a un **SUT**.

Si hay varias situaciones que se dean contemplar del mismo **SUT**, se puede emplear **when**.

Una vez descrito el enunciado, se finaliza con un bloque **in** y dentro la asercion

```
import org.scalatest.WordSpec

class SetSpec extends WordSpec {

  "A Set" when {
    "empty" should {
      "have size 0" in {
        assert(Set.empty.size == 0)
      }

      "produce NoSuchElementException when head is invoked" in {
        assertThrows[NoSuchElementException] {
          Set.empty.head
        }
      }
    }
  }
}
```

FreeSpec

Se define una clase que extienda **FreeSpec**, definiendose una serie de enunciados terminado por **in**, estos enunciados, se pueden partir para su reutilizacion empleando -

```
import org.scalatest.FreeSpec

class SetSpec extends FreeSpec {

  "A Set" - {
    "when empty" - {
      "should have size 0" in {
        assert(Set.empty.size == 0)
      }

      "should produce NoSuchElementException when head is invoked" in {
        assertThrows[NoSuchElementException] {
          Set.empty.head
        }
      }
    }
  }
}
```

Matchers

El API proporciona un **Trait** que proporciona numerosas funciones para poder realizar los asertos de forma mas semantica **org.scalatest.Matchers**, este emplea el verbo **should**, de forma paralela existe otro **Trait org.scalatest.MustMatchers** que emplea el verbo **must** son iguales, simplemente se emplea el segundo de forma mas formal.

La idea será sustituir los asertos por estas sentencias mas legibles

```
class ConMatchers extends FunSuite with Matchers {  
  
  val nimoy = new Person("Leonard Nimoy", 82)  
  val nimoy2 = new Person("Leonard Nimoy", 82)  
  val shatner = new Person("William Shatner", 82)  
  val ed = new Person("Ed Chigliak", 20)  
  
  test("nimoy == nimoy") { nimoy shouldBe nimoy }  
  test("nimoy == nimoy2") { nimoy shouldBe nimoy2 }  
  test("nimoy2 == nimoy") { nimoy2 shouldEqual nimoy }  
  test("nimoy != shatner") { nimoy should not be shatner }  
  test("shatner != nimoy") { shatner should not equals nimoy }  
  test("nimoy != null") { nimoy should not equals null }  
  test("nimoy != String") { nimoy should not equals "Leonard Nimoy" }  
  test("nimoy != ed") { nimoy should not equals ed }  
  test("nimoy es Person") {nimoy shouldBe a [String]}  
}
```

Existen muchos **Matcher**, cubriendo aspectos como

- La igualdad

```
result shouldEqual 3
```

- EL tamaño

```
result should have length 3
```

- Tratamiento de Strings

```
string should startWith ("Hello")
```

- Mayor o menor que

```
one should be < 7
```

- Tipologias

```
result1 shouldBe a [Tiger]
```

- Rangos

```
sevenDot0h should equal (6.9 +- 0.2)
```

- Vacio

```
traversable shouldBe empty
```

- Contenido

```
List(1, 2, 3) should contain (2)
```

- Agregaciones

```
List(1, 2, 3, 1) should contain only (1, 2, 3)
```

- Secuencias

```
List(1, 2, 2, 3, 3, 1) should contain inOrderOnly (1, 2, 3)
```

Ejecucion

Existen varias formas de ejecutar los test

- Con la consola de **Scala**

```
scala> run(new Test)
```

- Por linea de comandos haciendo referencia al **Runner** de Scalatest

```
$ scala -cp scalatest-RELEASE.jar org.scalatest.run ExampleSpec
```

- O con **sbt**

```
sbt> test
```

Implicitos

El API de Scala define la palabra reservada **implicit**, la cual se puede aplicar a:

- valores
- métodos
- clases

Valores Implícitos

Permiten extraer de la definición de un método el valor por defecto que tomará alguno de sus parámetros.

Por un lado se define en el contexto (ámbito) del método el valor implícito

```
implicit val valorParametroImplicito: Int = 0
```

Y por otro se define el método que haga uso de dicho valor implícito, para ello, se ha de separar los valores implícitos de los obligatorios en dos bloques de parámetros diferenciados, siendo el de implícitos el segundo bloque.

```
def metodo(parametroNormal: Int)(implicit parametroImplicito: Int): Int =  
    println (parametroNormal + "-" + parametroImplicito)
```

Solo es necesario poner la palabra reservada **implicit** al comienzo del bloque.

```
def metodo(parametroNormal: Int)(implicit parametroImplicito: Int,  
    otroParametroImplicito: String): Int =  
    println (parametroNormal + "-" + parametroImplicito + "-" + otroParametroImplicito)
```

El valor implícito se resuelve por tipo, luego si hubiese dos valores implícitos de tipo Int en este caso, se produciría un error de compilación de ambigüedad, ya que el método no sabría cual usar, de forma analoga, no se pueden definir dos parámetros implícitos en un mismo método con la misma tipología,

aunque si se necesita se puede recurrir a las **Clases Case**, creando un nuevo tipo que envuelva la tipologia deseada.

```
case class Path(s: String) extends AnyVal
case class BaseUrl(s: String) extends AnyVal

def foo(implicit a: Path, b: BaseUrl) = a.s ++ b.s

implicit val a: Path = Path("a")
implicit val b: BaseUrl = BaseUrl("b")
```

Métodos Implícitos

Son métodos que se definen en un determinado ambito que se ofrecen al compilador, como capaces de solventar referencias no especificadas

Entre otras cosas, permiten realizar conversiones entre tipos

```
//Tipologia que pretende extender las características del tipo *Int* de Scala
class MyInteger(i: Int) {
    def myNewMethod = println("hello from myNewMethod")
}

//Wrapper implícito, que permite convertir un Int en un MyInteger, para permitir que se
puedan invocar funcionalidades de MyInteger desde un Int
implicit def int2MyInt(i: Int) = new MyInteger(i)

//Se esta empleando de forma implícita la conversion a MyInteger
1.myNewMethod
```

Clases Implícitas

Permiten extender funcionalidades de clases ya definidas, envolviendolas de forma implícita

```

class ClaseExistente {
  def method1(): Int = 1
  def method2(n: Int): Boolean = true
}
implicit class ClaseImplicitaConMetodosExtra(claseExistente: ClaseExistente){
  def method3(n: Int): Boolean = false
  def method4(): Int = 0
  def method5(): Int = claseExistente.method1() * 2
}

val instanciaDeClaseExistenteImplicitamenteDotadaDeMasMetodos = new ClaseExistente

instanciaDeClaseExistenteImplicitamenteDotadaDeMasMetodos.method1()
instanciaDeClaseExistenteImplicitamenteDotadaDeMasMetodos.method2(5)
instanciaDeClaseExistenteImplicitamenteDotadaDeMasMetodos.method3(2)
instanciaDeClaseExistenteImplicitamenteDotadaDeMasMetodos.method4()
instanciaDeClaseExistenteImplicitamenteDotadaDeMasMetodos.method5()

```

NOTE | Ojo!!! No se pueden sobrescribir implementaciones de la clase que se esta vitaminando

Futuros

Se emplean para realizar operaciones costosas en segundo plano sin que se bloquee el hilo de ejecución principal.

Se basan en la clase **scala.concurrent.Future**, que permite encapsular la tarea de larga duracion, para que se ejecute en otro hilo

```

val f: Future[List[Int]] = Future {
  tareaDeLargaDuracionQueRetornaUnListDeEnteros
}

```

La clase **Future** es generica, teniendo que indicar el tipo de dato retornado por la tarea de larga duracion, en este caso **List[Int]**

Necesitan de un contexto de ejecucion, donde se definan las caracteristicas del **pool de thread** a emplear por los **Futuros**, el API define uno por defecto, que si se desea emplear, habrá que importar

```

import scala.concurrent.ExecutionContext.Implicits.global

```

Este **pool de thread**, basado en **ForkJoinPool** de Java, define un numero de threads igual al numero de procesadores disponibles para la ejecucion, lo cual se define con la caracteristica

Runtime.availableProcessors, esta propiedad puede ser modificada con el atributo de la maquina virtual **scala.concurrent.context.minThreads**.

Hay dos formas de ejecución de los Futuros:

- Una **sincrona**, que parará la ejecución del hilo desde donde se lance, hasta que termine la ejecución de la tarea envuelta por el **Futuro**, cosa poco interesante a priori, ya que sino no emplearíamos los futuros.

```
Await.result(f,Duration(2.0, TimeUnit.SECONDS))
```

- Otra **asincrona**, que se basará en **callbacks** que se ejecutará de forma paralela. Los posibles **callbacks** a definir son:
 - onComplete

```
f onComplete {  
  case Success(list) => print(list)  
  case Failure(e)    => println(e.getMessage)  
}
```

- onSuccess

```
f onSuccess {  
  case list => for (item <- list) println(item)  
}
```

- onFailure

```
f onFailure {  
  case e => println(e.getMessage)  
}
```

Se puede definir en un formato reducido

```
val f = Future {  
  tareaDeLargaDuracionQueRetornaUnListDeEnteros  
} andThen {  
  case Success(list) => println(list)  
  case Failure(e) => println(e.getMessage)  
}
```

Se puede simular una tarea de larga duracion durmiendo el hilo

```
def tareaDeLargaDuracionQueRetornaUnListDeEnteros: List[Int] = {  
  Thread.sleep(4000)  
  List(1, 2, 3)  
  //throw new Exception("Error al procesar la tarea de larga duracion")  
}
```

SBT

Emplea la estructura de directorio de **Maven**

Emplea el paradigma de **convention over configuration**

MAs informacion en la documentacion [aquí](#)

Comandos

clean → Limpia la carpeta **target**

compile → Compila los ficheros del proyecto y los coloca en la carpeta target

package → Crea un jar con la aplicación.

run → Ejecuta la aplicación

reload → Recarga la configuracion de SBT

plugins → Lista los plugins definidos

test → Ejecuta los test

test-only → Ejecuta solo los test que se le pasan separados por espacios

new → Permite acceder a las plantillas de [gitter8](#), para la creación de aplicaciones preformadas.

Algunas de estas plantillas se pueden encontrar [aquí](#)

Un ejemplo de creación de proyecto basico Scala con plantilla

```
sbt new ferhtaydn/sbt-skeleton.g8
```

Key

Todas las definiciones son **Keys**, pueden ser:

- `SettingKey[T]` → una key para un valor que se calcula una sola vez (el valor es calculado cuando se carga el proyecto, y se mantiene).
- `TaskKey[T]` → una key para un valor, llamado una task (tarea), que tiene que ser recalculada cada vez, potencialmente con efectos laterales.
- `InputKey[T]` → una key para una task que tiene argumentos para la línea de comandos como entrada.

La definición de una key se escribe

```
name := "hello"
```

La sentencia se puede interpretar como

```
name.:=("hello")
```

Es necesario que exista entre medias de cada definicion una linea en blanco.

Keys predefinidas

Las principales Keys que deben aparecer en un proyecto **SBT** son

```
name := "Simple Project"

version := "1.0"

scalaVersion := "2.10.4"
```

Custom Key

Se pueden declarar nuevas **Task** empleando los metodos de creación: `settingKey`, `taskKey` e `inputKey`

```
lazy val hello = taskKey[Unit]("An example task")
```

Y definir su valor de forma analoga a las proporcionadas por la herramienta

```
hello := { println("Hello!") }
```

Plugins

Los plugins añaden nuevas **Key**, típicamente **TaskKey**.

Se definen en el fichero `/project/plugins.sbt`

Por ejemplo se puede añadir un Plugin para transformar un proyecto **SBT** a **Eclipse**

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "4.0.0")
```

Este plugin da un nuevo **TaskKey** llamado **eclipse** que interpreta la configuración **SBT** y genera los ficheros de **eclipse**, si se hace alguna modificación en la configuración de **SBT**, habrá que volver a lanzar el comando para que eclipse la tenga en cuenta.

Si el plugin no está en los repositorios por defecto, se pueden definir nuevos

```
resolvers += Resolver.sonatypeRepo("public")
```

El listado de plugins disponible se puede consultar [aquí](#)

Dependencias

Las dependencias se añaden sobre la **Key libraryDependencies**, en este caso es una **Key** que no tiene un único valor, por lo que se emplea el operador `+=` para concatenar las distintas dependencias

```
libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3"  
libraryDependencies += "org.scalatest" % "scalatest_2.11" % "2.2.4" % "test"
```

El valor se obtiene empleando el método `%` que permite construir un ID para un módulo de **Ivy**.

Se puede obtener la referencia de la dependencia de [aquí](#)

NOTE

Hay dependencias que están creadas específicamente para una versión de Scala concreta, como es el caso de `scalatest`, en el ejemplo se está empleando la versión concreta para Scala 2.11 (**scalatest_2.11**), para no tener que estar cambiando estas referencias al cambiar la versión de Scala, se permite la sintaxis

```
libraryDependencies += "org.scalatest" %% "scalatest" % "2.2.4" % "test"
```

Donde la aparición de `%%` lo que hace es añadir el valor de la propiedad **scalaVersion** a continuación del **artifactId**

Ambitos

Se pueden definir los valores de los **key** para los distintos ambitos definidos

- Test
- Compile
- Runtime
- Global

Siguiendo la sintaxis

```
name in Test := "HolaMundoSBTest"
```

Desde la linea de comando se puede inspeccionar los valores que tienen las distintas **key**

```
sbt> inspect test:name
```

Actores

Son otra forma de gestionar la concurrencia en **Scala**.

Los **Actores** son componentes con estado que procesan en un hilo distinto, pero de manera **secuencial** una cola de mensajes, asociando (o no) a cada uno de estos mensajes un cierto algoritmo.

Actualmente se esta incluyendo el API de **Akka Actors** dentro de **Scala**, pero de momento hay que incluirlo como dependencia, la documentacion [aquí](#)

Para incluir el API de **Akka Actors** con **SBT**

```
libraryDependencies += "com.typesafe.akka" %% "akka-actor" % "2.5.16"
```

Para definir un **Actor** con este API hay que heredar del Trait **akka.actor.Actor**, que obliga a implementar el método **receive**, donde se procesan los mensajes

```
class HelloActor extends Actor {  
  def receive = {  
    case "hello" => println("hello back at you")  
    case _ => println("huh?")  
  }  
}
```

Para instanciarlo, se hace a través de una instancia de la clase **ActorSystem**

```
val system = ActorSystem("HelloSystem")  
  
val helloActor = system.actorOf(Props[HelloActor], name = "helloactor")
```

El método **actorOf** retorna una referencia a un objeto **ActorRef**

El segundo parámetro del método **actorOf** es opcional, permite dar un nombre al actor de cara a monitorizar su estado.

En el anterior ejemplo, se está creando un **Actor** que no define propiedades de construcción, si el actor las definiese, la sintaxis sería

```
val myActor = system.actorOf(Props(new HelloActor("...")), name = "myactor")
```

Para enviar mensajes, se emplea el método **!** del objeto **ActorRef**

```
helloActor ! "hello"  
helloActor ! "buenos días"
```

Para parar el sistema de Actores, se ha de lanzar el comando **terminate**

```
system.terminate()
```

Si se desea parar algún actor en concreto dentro del sistema

```
system.stop(helloResponseActor)
```

Respuestas

Los **Actores** pueden responder al cliente que lanzó la ejecución, pero dado que no realizarán la ejecución en el mismo hilo, se precisarán de **Futuros**, lo cual significa que hay que definir el contexto

```
import scala.concurrent.ExecutionContext.Implicits.global
```

Para realizar la petición y definir un callback que procese la respuesta, la petición se ha de hacer con **akka.pattern.ask**, la cual tiene un parámetro implícito de tipo **Timeout**

```

val system = ActorSystem("HelloResponseSystem")
val helloResponseActor = system.actorOf(Props[HelloResponseActor], name =
"helloResponseActor")

implicit val timeout = Timeout.apply(2, TimeUnit.SECONDS)

val future = ask(helloResponseActor, "hello")
future onComplete {
    case util.Success(mensaje) => println(mensaje)
    case util.Failure(e)      => println(e.getMessage)
}

```

La respuesta desde el **Actor**, se realizará a través del objeto **sender**, definido por el API en todos los **Actor**, que es un **ActorRef** y por tanto tiene el método !

```

class HelloResponseActor extends Actor {
    def receive = {
        case "hello" => { println("hello back at you"); sender ! "continue" }
        case _       => println("huh?")
    }
}

```