Scala

Victor Herrero Cazurro

Contenidos

Introduccion

Scala es

- Un lenguaje de programacion moderno creado por **Martin Odersky**, influenciado por Java, Ruby, Smalltalk, ML, Haskell, Erlang y otros.
- Un lenguaje de programacion orientado a objetos puro, donde toda variable apunta a un objeto, no existen primitivos y todas las operaciones son métodos.
- Un lenguaje de programación funcional, donde las funciones son referenciables por variables y se pueden pasar por parametros.
- Un lenguaje que corre en la JVM, por lo que se puede emplear todas las librerias que esta incluye.

Algunas caracteristicas

- No se necesita terminar las sentencias con **;**, solo se empleará cuando haya que seperar sentencias en la misma linea, dado que el salto de linea es tambien identificar de fin de sentencia.
- Se puede prescindir de as {} en bloques de codigo mono-sentencia.
- En la ejecución de funciones se puede prescindir de los () si solo se pasa un parametro.
- Se puede prescindir del operador **.** para acceder a los miembros de una instancia.
- Es **Case Sensitivity**, los identificadores **Hello** y **hello** son diferentes..
- El nombrado de las clases debe empezar en mayusculas y si el nombre es compuesto, cada una de las palabras que lo forme, tambien debe empezar en mayusculas.
- El nombrado de los métodos debe empezar en minusculas y si el nombre es compuesto, cada una de las siguientes palabras que lo forme, tambien debe empezar en mayusculas.
- El nombre del fichero debe ser exactamente igual al Objeto que se define dentro.
- El método de entrada a la ejecución es

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

- El resultado de la ejecución de un método, será el resultado de la ejecución de la última sentencia, no se emplea **return**.

Beneficios

- Conciso
- Legible
- Expresivo (no verboso)

Veamos un ejemplo donde se puede observar la diferencia entre Scala y Java, se trata de un ejemplo

de como se puede aplicar un filtrado de elementos sobre una coleccion.

```
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 pagínas donde ejecutar codigo Scala online, como scalafiddle, scalakata o scastie

Compilación

Con el SDK, se proporciona el compilador **scalac**

```
scalac HelloWorld.scala
```

Scala y Java

Scala es 100% compatible con el codigo 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.

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]

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 unicos asociados a un Singleton.

NOTE	La definición de este **object** debe estar en un fichero con el nombre del **object**, en este caso **Programa.scala**, aunque tambien se definan otras clases o **object** dentro

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!!!!")
}
```

Clases

Una **clase** en escala, 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(0, "")
    }
}
```

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.

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.

```
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

```
def x() : String = {
    return this.x
}

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
}
```

Inicializacion de Campos con Bloques de Codigo

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"
  }
}
```

NOTE

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

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

Tipo Option

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

- **None** permite indicar en escala nada.
- **Some(T)** permite encapsular cualquier tipo de 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 obtuviesemos el valor real, tendriamos 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")
}
```

Equals y HashCode

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

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

Se puede aprovechar **Pattern Matching** para implementarlo

```
override def equals(that: Any): Boolean = that match {
  case that: Person => that.isInstanceOf[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 deve 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]
```

Objetos

Para crear una concrecion de una **clase**, se emplea la palabra reservada **new**

```
new Persona
```

En Scala no existen 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, dado que la construcción del objeto es automatica y gestionada por **scala**, por lo que no habria posibilidad de cambiar dichos parametros, seria constantes.

Objeto Compañia

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

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

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 cree 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
```

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.

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 tipologia **Int**, no existe el concepto del operador y se pueden nombrar funciones con caracteres especiales como +, -, ... que en otros lenguajes estan reservados como operadores, podemos decir que los operadores se pueden sobreescribir.

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.

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

Null → Referencia vacia o null.

Nothing → Subtipo de todos los demas incluidos los Null.

Any → Supertipo de todos los tipos (equivalente a **Objetc** en java).

AnyRef → Superttipo de los tipos referencia (variables).

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 cientifica (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 → \u000c → 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
vivia el ilustre hidalgo Don Quijote..."""
```

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

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 cualquie 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 ;-)
         }
      }
   }
}
```

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
```

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")
}
```

For

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

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

En el ejemplo la funcion 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 iteracion.

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

Contadores

Por ejemplo, **until** se suele emplear para recorrer la secuencia empleando **Contadores**, para conocer la posición del elemeno en a secuencia

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

Se pueden definir varioss contadores, produciendose la ejecución del buble con todas las conbinaciones 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")
```

Rangos

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)))
```

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 procesa 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**

NOTE

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

```
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 encntrado 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 desa procesaro el valor 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;
}
```

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, de hacerlo, retornarán el valor de la última sentencia ejecutada, que no puede ser de asignación.

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
    }

}
```

En Scala la invocación de las funciones asociadas a variables, no se tiene porque escribir con el operador **.**

Por tanto la expresion

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

Es equivalente a

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

Funcion anonima

Se pueden definir funciones anonimas como la del siguiente ejemplo

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

De formas mas reducida

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

Aplicando un funcion naonima en un trozo de programa.

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

    def main(args: Array[String]) {
        oncePerSecond(() => println("time flies like an arrow..."))
    }
}
```

Empleando la siguiente sintaxis en la declaración de la función anonima

```
<variable que referencia a la función que llega por parametros>: (<parametros
separados por ,>) => <tipo retornado>
```

Donde **callback** es el parametro que apuntará a la función recibida y se observa que la declaración de la función anonima que recibira la función **oncePerSecond** sigue la sintaxis

```
() => Unit
```

Que representa la firma de cualquier función que no recibe parametros ni retorna valores, por tanto **Unit** es similar a **void** en Java.

Y 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

```
() => println("time flies like an arrow...")
```

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 cuanquier orden, empleando los nombres de los parametros para su asignacion.

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

Numero indeterminado de patametros

Se puede definir un método para que acepte unn 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

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")
```

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 dias, " + 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 **boleano** 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, sino con las de otro ambito.

```
object AplicacionClosures extends App {

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

  //Se define un metodo, que usa la anterior variable
  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**.

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**

Traits (FP)

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

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

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 implmenta 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 algun **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
    }
}
```

Clases Abstractas

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

- Cuando quieras crear una clase con parametros de construcción.
- El codigo tenga quer 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"
}
```

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 {
        }
    }
}
```

Imports

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

- Agrupación de import de un paquete

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

- Importación de todas las clase de un paquete

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

- Renombrado de un import

```
import scala.collection.{Vector => Vec28}
```

- 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("")
    }
}
```

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.

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

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 predefenidas

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**, tipicamente **TaskKey**.

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

Por ejemplo se puede añadir un Plugin para tranformar 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 configuracion **SBT** y genera los ficheros de **eclipse**, si se hace alguna modificación en la condfguracion de **SBT**, habra que volver a lanzar el comando para que eclipse la tenga en cuenta.

Si el plugin no esta 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 unico valor, por lo que se emplea es el operador `+=` para concatenar las distitnas 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 aqui

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
```

Testing

El la Programación funcional, el testing de las funcionalidades, ofrece mayor garantia que en la programacion imperativa, dado que en la funcional el resultado de la ejecución de una función, solo dependen de los parametros de entrada, y no del ambito de ejecución.

El Framework para la realizacion de test en **Scala** se llama **scalatest**, se puede encontrar su documentacion aqui

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 añadirlo con **Maven**

```
<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
- FlatSpec
- FunSpec
- WordSpec
- FreeSpec
- PropSpec
- FeatureSpec

FunSuite

Se define una clase que extienda **FunSuite**, donde se definirá la logica del **Test** y se incluiran 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** siguen definen un texto descripivo del test 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**

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

```
sevenDotOh 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
```

Clases Case (FP)

Un ejemplo de definición

```
abstract class Arbol
case class Sum(l: Arbol, r: Arbol) extends Arbol
case class Var(n: String) extends Arbol
case class Const(v: Int) extends Arbol
```

Difieren de las clases normales en

- No es obligatorio utilizar la palabra clave **new** para crear instancias de estas clases.

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

- Los parametros del constructor son considerados campos **val**:
- Se proveen por defecto los siguientes métodos
  - 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**
  - accessor → ya que por defecto los parametros del constructor son considerados **val**, y en el caso de definirse como **var**, tambien 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.

```
p match {
    case Person(n) => println(n)
}
```

Algunos ejemplos de patrones a emplear

```
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(",")}"
}
```

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 compañia** (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)
}
```

Sentencias Case condicionales (Guards)

Se pueden añadir sentencias **if** a los case en un **match**

```
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")
}
```

Colecciones

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

- Las hay mutables e inmutables, aunque de las inmutables hay alias y no se necesita añadir **imports**
- Predicado
- Funcion anonima
- Bucles implicitos

Veamos un ejemplo de uso de una **funcion predicado**, definida como **funcion anonima** y aplicada a un método de una coleccion como es **filter**, que internamente recorre los elementos de la colecion con un **bucle implicito** para generar una nueva coleccion.

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

//numeros = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

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

//pares = List(2, 4, 6, 8)
```

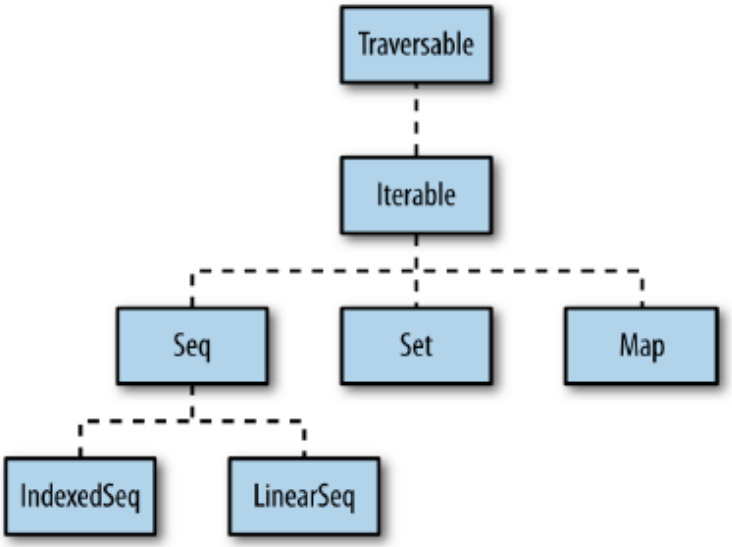
El bucle inplicito en **filter** seria algo como

```
for {
    item <- list
    if item % 2 == 0
} yield item
```

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 vsision general, no lleva a los principales **Traits**

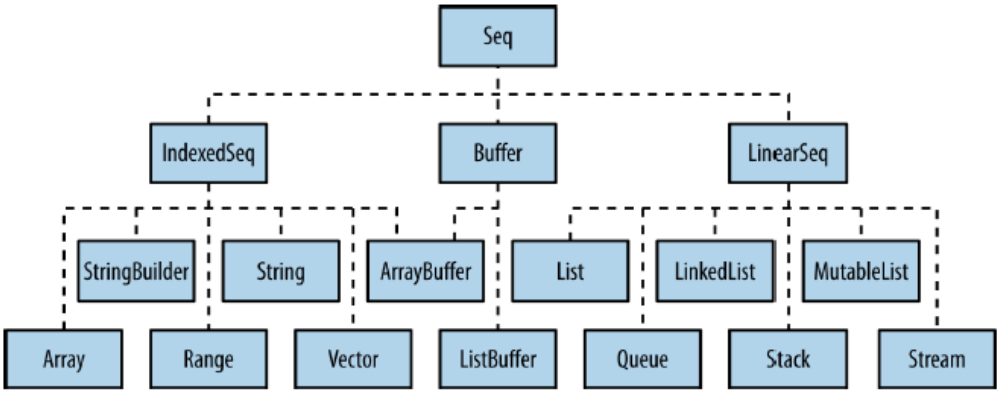


Como se ve el elemento de mayor jerarquia es **Transversable**, siendo la implementacion por defecto **List**

```
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** y **tail**. 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

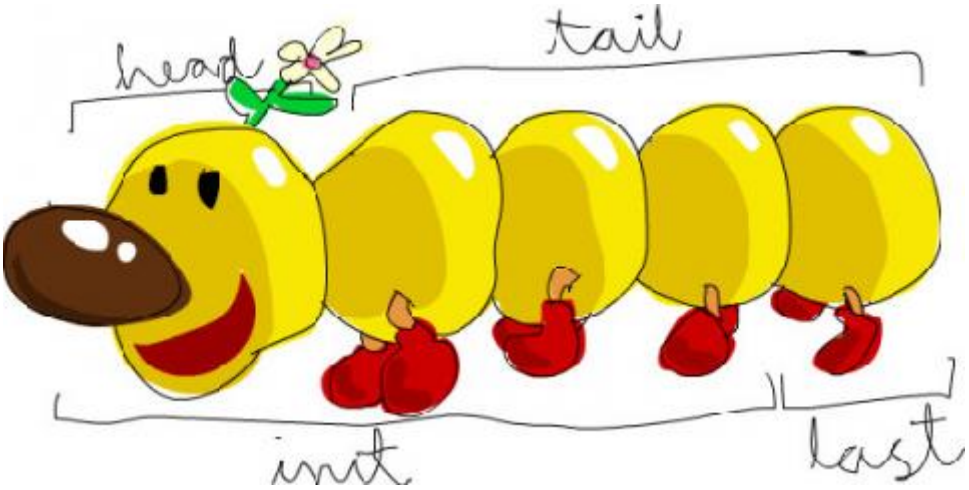
```
var nums = ArrayBuffer(1, 2, 3)
//nums = scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 3)
nums += 4
//nums =  scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 3, 4)
```

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

- Range → Rango de valores enteros (inmutable)
- String → secuencia de caracteres indexada (inmutable)
- Arrray → Array de Java, que puede cambiar sus elementos, pero no de tamaño (mutable)
- ArrayStack → Estructura LIFO que mejora el rendimineto 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 estan pensadas para la recursividad.



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

Las implementaciones de referencia son

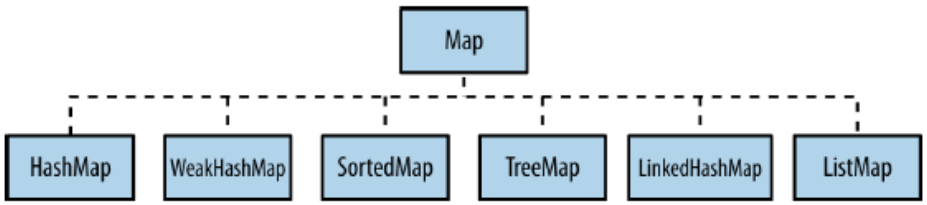
- List → Inmutable
- ListBuffer → Mutable

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 sequencias muy largas. (inmutable)
- DoubleLinkedList → Como **LinkedList** pero bidireccional (mutable)
- LinkedList → Como **List** pero mutable.

Mapas

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



Tambien los hay mutables e inmutables

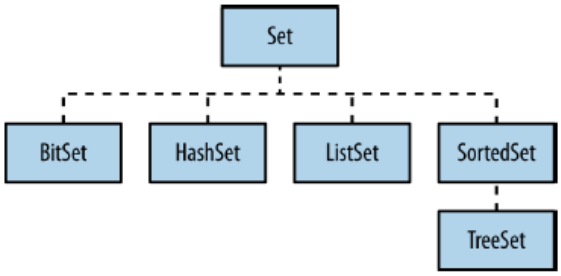
```
val m = Map(1 -> "a", 2 -> "b")

//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 implementacion (mutable / inmutable)
- LinkedHasMap → Retorna los elementos en orden de insercion (mutable)
- ListMap → Internamente emplea un LinearSeq. Los elementos son insertados en la cabecera de la coleccion, por lo que los retorna en orden inverso a la insercion (mutable / inmutable)
- SortedMap → Igual que **TreeMap**
- TreeMap → Implementacion de arbolo rojo-negro (inmutable)
- 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 / inmutable)
- HashSet → Emplea **HashTree** y **HashTable** para su implementacion (mutable / inmutable)
- LinkedHasSet → Retorna los elementos en orden de insercion (mutable)
- ListSet → implementado con un List (inmutable)
- 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 precesos, 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 direfencia 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 collecion 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→ Retorna nacoleccion con todos los elementos que cumplen el predicado.

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

Un ejemplo de uso con mapas

```
val extensions = Map("steve" -> 100, "bob" -> 101, "joe" -> 201)
extensions.filter((namePhone: (String, Int)) => namePhone._2 < 200)
// scala.collection.immutable.Map[String,Int] = Map((steve,100), (bob,101))

//Empleando patter matching
extensions.filter({case (name, extension) => extension < 200})
res0: 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 tranformación:
- += 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 sonbre 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→ Retorna un nueva coleccion aplicando una funcion a todos los elementos de la coleccion
- 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 coleccion con los elementos ordenados segun la funcion definida, esta funcion, tendra dos parametros 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, valido 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 coleccion 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 coleccion 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 indice de la coleccion, creando las tuplas como (elemento, posicion)

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

- Métodos de agrupación:
  - groupBy p → Retorna un mapa con dos colecciones, donde una coleccion 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 coleccion 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 coleccion 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 funcion predicado
  - endsWith
  - exists → retorna True si el predicado se cumple para algun elemento
  - forAll p → retorna True si el predicado se cumple para todos los elementos
  - hasDefiniteSize → Retorna true si la coleccion tiene un tamaño definido, retorna false para los **Stream**
  - indexOf →
  - indexOfSlice
  - indexWhere
  - isDefinedAt

    - isEmpty → Retorna true si la coleccion esta vacia.
    - lastIndexOf
    - lastIndexOfSlice
    - lastIndexWhere
    - max → Retorna el mayor elemento de la coleccion
    - min → Retorna el menor elemento de la coleccion
    - nonEmpty → Retorna true si la coleccion no esta vacia
    - product → Retorna la multiplicacion de todos los elementos de la coleccion
    - segmentLength
    - size → retorna el tamaño de la coleccion
    - 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)
x.foreach(i => println(i))
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 elminado el elemento con clave k
  - - (k1, k2, k3) → Retorna un nuevo mapa, donde se han elminado 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 situacion, añadir elementos sobre una coleccion 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 coleccion sea inmutable, sino que se estan creando nuevas coleccion 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 podria realizar es

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

Iterators

Son un tipo de coleccion, 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 transformacion a otro tipo de coleccion con los métodos: toArray, toBuffer, toIndexedSeq, toIterable, toIterator, toList, toMap, toSeq, toSet, toStream, toString y toTraversable.

Arrays

Los arrays se define 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

tipologia.

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 tipologia.

Permite por ejemplo manejar mas de un objeto sin tener que definir una tipologia 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
```

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 {
    tareaDeLargaDuracion
}
```

La clase **Furute** 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 habra que declarar

```
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 priorí, 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)
}
```

- onFailiure

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

Se puede definir en un formato reducido

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

Se puede simular una tarea de larga duracion durmiendo el hilo

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

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_2.11" % "2.4.17"
```

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

Een el anterior ejemplo, se esta 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 dias"
```

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

```
system.terminate()
```

Si se desea parar algun actor en concreto dentro del sistema

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

Respuestas

Los **Actores** pueden responder al cliente que lanzo la ejecución, pero dado que no realizarán la ejecución en el mismo hilo, se precisarán de **Futuros**.

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 parametro implicito 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 Success(mensaje) => println(mensaje)
    case 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?")
    }
}
```

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 pertener 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]), puediendo 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]), puediendo 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
```

Conversiones

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

Implicitos

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

- valores
- métodos
- clases

Valores Implicitos

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

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

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

Y por otro se define el método que haga uso de dicho valor implicito

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

NOTE	Ojo!!! que el valor implicito se resuelve por tipo, luego si hubiese dos valores implicitos de tipo Int en este caso, se produciria un error de compilación de ambiguedad

Métodos Implicitos

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 caracteristicas del tipo *Int* de Scala
class MyInteger(i: Int) {
    def myNewMethod = println("hello  from  myNewMethod")
}

//Wrapper implicito, 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 implicita la conversion a MyInteger
1.myNewMethod
```

Clases Implicitas

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

```
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

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 pefijo **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 propoprciona **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")
```

Extractores

Son **Objectos** de Scala que implementan los métodos **apply** y **unapply**, de forma analoga a como lo hacen las clases case, pero sin serlo.

Son empleados junto con el **Pattern Mattching**

```
object Twice {
  def apply(x: Int): Int = x * 2
  def unapply(z: Int): Option[Int] = if (z%2 == 0) Some(z/2) else None
}
object TwiceTest extends App {
    val x = Twice(21) //Aplica apply
    println(x) //x = 42
    x match {
        case Twice(n) => { //Aplica unapply
            println(n) //n = 21
        }
    }
}
```

Resumen Programacion Funcional

En programacion funcional

- **lambda** es equivalente a **funcion**.
- Las variables son inmutables (final)
- Las funciones no pueden modificar o verse afectadas por un estado global, por lo que se puede paralelizar la ejecución de las sentencias.
- **currying**, patrón basado en el patrón adaptador. 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 muchos casos el **curriying** lleva a que en una determinada funcion que acepta **n** parametros de entrada algunos de ellos son constantes, y al adaptarla, se define una nueva funcion que absorbe dichas constantes

```
def elCuadradoDe(i : Int) : Int = {
    scala.math.pow(i,3)
}

elCuadradoDe 1;
```

Si aplicamos la caracteristica de **Scala** de **Invocacion Parcial**, simplificariamos la anterior expresion

```
var elCuadradoDe = scala.math.pow(_: Double, 2.toDouble)
```

Donde estamos dejando abierto el valor del primer parametro, pero cerrando el del segundo a **2**

Scala permite la definicion **currying** de otra forma

```
def exponencial(x: Double) = (y: Double) => scala.math.pow(x, y)
```

En este caso lo que vemos es que la primera funcion, que acepta un parametro **x**, retorna otra funcion que acepta un parametro **y**, la cual no se podra invocar hasta que se proporcione dicho parametro.

La invocacion se puede hacer por pasos

```
var funcionParcial = exponencial(7)
funcionParcial(2)
```

O directamente, que quedaria mas legible

```
exponencial(7)(2)
```

- **Funcion De Orden Superior** → Aquellas funciones que reciben por parametros otras funciones para invocarlas internamente
- **Closures** → Función que puede interaccionar no solo con las variables de su ambito, sino con las de otro ambito, sucede al emplear **funciones de orden superior**, donde la funcion recibida, que es ejecutada desde la **funcion de orden superior**, puede tener acceso a variables de un ambito distinto a la **funcion de orden superior**

```
object Aplicacion extends App {
  Ambito2.funcionDeOrdenSuperior(Ambito1.sayHello, "Pedro")
}

object Ambito1 {
  //Se define una variable
  private var hello = "Hello"

  //Se define un metodo, que usa la anterior variable
  def sayHello(name: String) { println(s"$hello, $name") }
}

object Ambito2 {
  def funcionDeOrdenSuperior(f: (String) => Unit, name: String) {
    f(name)
  }
}
```

- **Pattern Matching** → Reconocimiento de patrones, que permite identificar un dato por el cumplimiento de una condicion. Se basa en los **Case**

```
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(",")}"
}
```

- **Evaluación tardia** → ejecucion del codigo solo cuando es necesario, no hay garatia del orden, e incluso alguna sentencia puede que nunca se llegue a ejecutar, porque su ejecución dependia del empleo de la variable asociada, que no fue empleada por que no se cumplio una condición.

Se traduce en variables **lazy**

```
lazy val perezosa = "perezosa"
```

O en colecciones perezosas **Stream**

- **concurrencia** → Dado que el resultado de la ejecucion de una funcion no depende del entorno, se pueden emplear las funciones de forma concurrente, dado que ninguna operacion que haga un hilo, dejara rastro que afecte a otro hilo.

Se traduce en **Futuros**

```
val f: Future[List[Int]] = Future {
    tareaDeLargaDuracion
}

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

Y **Actores**

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

object Simple extends App {
  val system = ActorSystem("HelloSystem")
  val helloActor = system.actorOf(Props[HelloActor], name = "helloactor")
  helloActor ! "hello"
}
```

- **Recursividad** → Invocacion de una funcion desde ella misma, camiando los parametros de entrada para evitar bucles infinitos.

```
def sum(xs: List[Int]): Int = {
    if (xs.isEmpty) 0
    else xs.head + sum(xs.tail)
}
```