Programación Funcional

J.A. Medina

Ciencias de la Computación

Universidad de Alcalá

Introducción

Objetivos

- Aprender a programar lenguajes funcionales y conseguir
  - Alto rendimiento
  - Funcionalidad y mantenibilidad
  - Escalabilidad
- Adquirir conocimiento técnico relativo a
  - Principios y patrones de programación funcional
  - Manejo de la APIs, sus herramientas y técnicas

Recursos en la Web

- Aula Virtual

  Código de ejemplo, transparencias empleadas en clase, ejercicios, planificación,…

- The Scala Programming Language

  Herramientas de desarrollo, documentación, código de ejemplo, videos, APIs,…

Bibliografía

1. **Odersky, Spoon & Venners, ”Programming in Scala – A Comprehensive Step-by-step Guide”, Artima, 2008**

2. Wampler & Payne, “Programming Scala”, O’Reilly, 2009

3. Odersky, M, “Scala By Example”, Programming Methods Laboratory – EPFL Switzerland

¿Qué es SCALA?

Scala es un lenguaje de programación general diseñado para expresar patrones de programación comunes de manera elegante, concisa y segura (con respecto a tipos).

- Paradigmas: **orientado a objetos** y **funcional**
- Se integra perfectamente con **Java**
  - de manera más clara que Clojure p.e.
- El compilador es muy **seguro**
  - escrito por M. Odersky, el mismo que escribió el compilador de referencia de Java
- Scala es **estáticamente tipado** (inferencia de tipos)
  - tipificación se realiza durante la compilación, y no durante la ejecución
- Scala es **extensible**
  - método puede ser usado como un operador, compatibilidad con Java y sus librerías
- Scala interopera con las plataformas **Java** y **.NET**

¿Quién usa SCALA?

theguardian
Winner of the Pulitzer prize

foursquare

Emplean Lift (un framework sobre Scala) en su servidor (tanto su website como su versión móvil y su API REST).

LinkedIn

Al menos las partes de LinkedIn Social Graph (65+ millones de nodos, 680+ millones de aristas y más de 250 millones de solicitudes por día.

twitter

Migraron (tras una gran catástrofe) su cola de mensajes de Ruby a Scala.

Novell

XEROX

SONY

SIEMENS

Y muchos más…

http://alvinalexander.com/scala/whos-using-scala-akka-play-framework

¿Qué es Lenguaje Funcional?

- Lenguajes funcionales cada vez mas demandados en el mercado laboral
- Los lenguajes funcionales más populares actualmente son Lisp, Haskell, Erlang, Scala y Clojure
- Aprender el lenguaje funcional nos puede brindar muchas ideas interesantes, patrones, buenas costumbres y lecciones que podemos aplicar a muchos otros lenguajes.
- Aprender un lenguaje bien, nos hace cambiar cómo pensamos y cómo resolvemos los problemas

¿Qué es Lenguaje Funcional?

Características comunes en lenguajes funcionales:

- No usar variables, sino “valores con nombre” o dicho de otra manera, evitar asignar más de una vez a cada variable
- La construcción if sigue el patrón funcional de ser una expresión que devuelve un valor. En este caso, recibe una condición y dos expresiones: la primera se devuelve si la condición es verdadera, y la segunda si la condición resulta ser falsa.

  Funciona de una forma parecida al operador ternario de C y Java

  (valor = condicion ? valorsiverdadero : valorsifalso).

¿Qué es Lenguaje Funcional?

Pensar en el objetivo, no en el proceso

Usar variables que no cambian una vez les hemos asignado un valor, es una buena costumbre porque hace más fácil entender de dónde sale cada valor.

Scala distingue entre dos tipos de variables: var y val.

El segundo tipo, que es el más usado con diferencia, declara una variable inmutable, por lo que el compilador nos asegura que no podemos asignar ningún otro valor una vez declarada.

Diseño de dentro a afuera, mediante funciones pequeñas

Escribir funciones y métodos de una o dos líneas es útil *si* suben el nivel de abstracción.

¿Qué es Lenguaje Funcional?

Efectos colaterales

Un efecto colateral es cualquier cambio que una función produce fuera del ámbito de la función en sí.

Una función que modifique una variable que ha recibido como parámetro o que modifique variables globales o cualquier otra cosa que no sean variables locales a la función está produciendo efectos colaterales

¿Qué es Lenguaje Funcional?

Funciones de orden superior

- Las funciones son valores más o menos normales que se pueden pasar como parámetros, asignar a variables y devolver como resultado de la llamada a una función

```
def sum(f: Int => Int): (Int, Int) => Int = {
def sumf(a: Int, b: Int): Int = {...}
sumf
}
def cube(x: Int) => x * x * x
sum(x => x * x * x)(1, 10) // sum of cubes from 1 to 10
sum(cube)(1, 10)
```

¿Qué es Lenguaje Funcional?

Evaluación perezosa

- Consiste en no hacer cálculos que no sean necesarios
- Se produce cuando escribimos una función que genere recursivamente una lista de 10 elementos, y otra función que llame a la lista pero que use el tercer elemento.
- Por lo que sólo ejecutará la función hasta que se *genere* el tercer elemento de la lista, y luego continuará con la ejecución del programa principal
- La evaluación perezosa proporciona una mayor eficiencia

Ejemplo de programación funcional

En matemáticas, se define el máximo común divisor (abreviado mcd) de dos o más números enteros al mayor número que los divide sin dejar resto.
Por ejemplo, el mcd de 42 y 56 es 14. En efecto: operando: 42/14= 3 y 56 /14 = 4.

```
def gcdLoop(x:Long, y:Long):Long={
        var a = x
        var b = y
        while (a != 0) {
              val temp = a
              a = b % a
              b = temp }
 b }
```

```
def gcd(x:Long, y:Long):Long={
        if (y == 0)
            x
        else
            gcd(y, x % y)
}
```

Pasos iniciales

PASO 1: El intérprete

El intérprete es una shell interactiva.

```
$ scala
Welcome to Scala version 2.7.2.
Type in expressions to have them evaluated.
Type :help for more information.

scala> 1 + 2

res0: Int = 3
```

Datos introducidos por el usuario

Resultado devuelto por Scala:

- `res0`: variable donde se guarda el resultado
- `Int`: tipo de la variable (Integer de Scala)
- `3`: resultado de la operación

PASO 1: El intérprete (y2)

Scala tiene los mismos tipos que Java:

- scala.Int → Java int
- scala.Boolean → Java boolean
- scala.Float → Java float
- ….

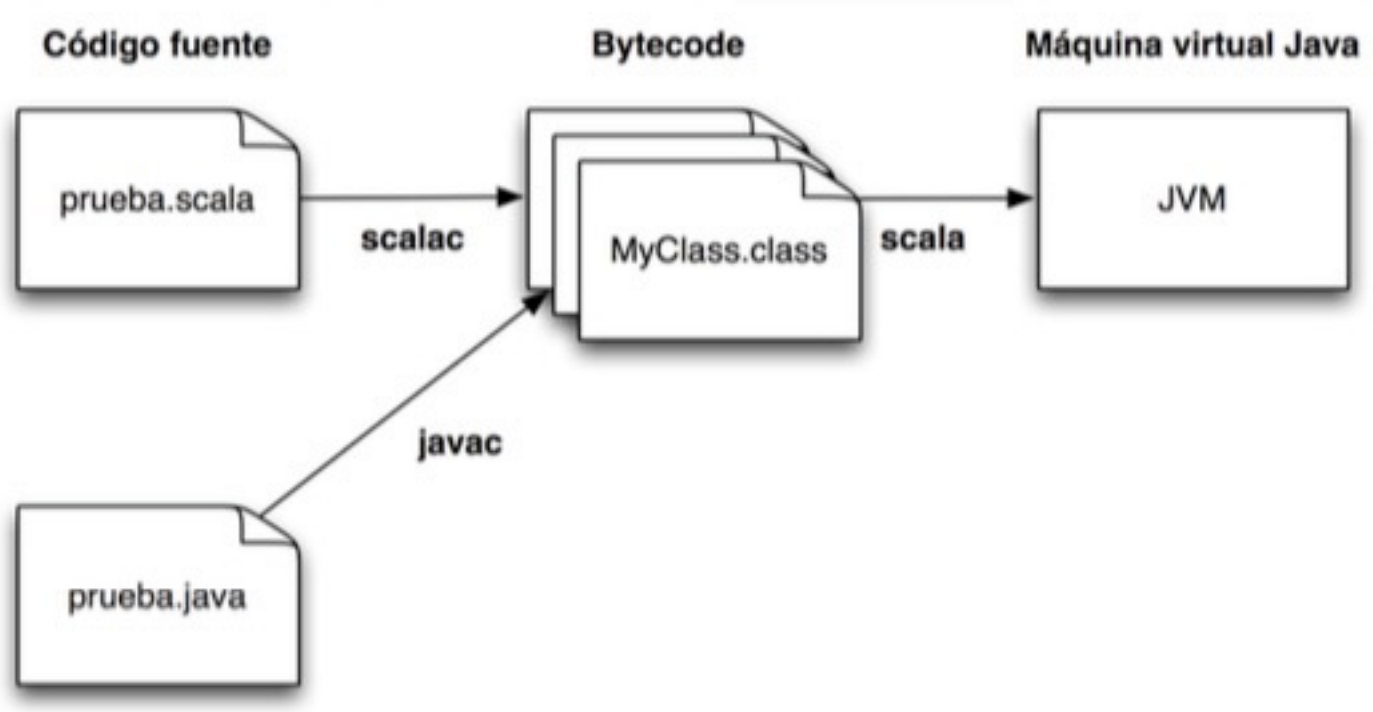
```
scala> res0 * 3

res1: Int = 9
```

Por ejemplo, para imprimir un mensaje en pantalla:

- println -> System.out.println (Java)

```
scala> println("¡Hola, mundo!")

¡Hola, mundo!
```

PASO 1: El intérprete (y3)




```
scala> scalac nombre archivo //compila
scala> scala nombre archivo  //ejecuta
```

Paso 2: Definición de variables

Dos tipos de variables:

- Val: similar a la variable de tipo final en Java

  Ej. `final double PI=3.141592653589793`
- Var: variable clásica.

```
scala> val msg = "¡Hola, mundo!"
msg: java.lang.String = ¡Hola, mundo!

scala> msg = "¡Adiós mundo cruel!"
<console>:5: error: reassignment to val
            msg = “¡Adiós mundo cruel!”
                ^
```

Usar
`var msg = …`

Paso 3: Definición de funciones

```
scala> def greet() = println("¡Hola, mundo!")
greet: ()Unit

def <nombre de la función>(Parametro1:tipo1>, …):<tipo_resultado>={
    <cuerpo de la función>}

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

scala> max(3, 5)
res6: Int = 5
```

- Scala permite definir funciones anónimas.
- Una función anónima es conocida como ***función literal***.

```
$ var duplica = (x:Int) => 2*x
```

Cuerpo función

Signatura función

```
val añadir = ( un : Int , b : int ) => a + b
añadir ( 1 , 2 ) // El resultado es 3
```

Paso 4: Scripts

```
/* Hola.scala */

println(“¡Hola, mundo, desde un script!”)
```

```
$ scala hola.scala

¡Hola, mundo, desde un script!
```

```
/* HolaParametro.scala */

println("¡Hola, "+ args(0) +"!”)
```

- Los arrays empiezan en `0`
- Emplean `()` y no `[]` como Java

```
$ scala hola.scala Juan

¡Hola, Juan!
```

Paso 5: Alternativas y repeticiones

```
/* EchoArgs.scala */
var i = 0
while  (i < args.length) {
      if (i  != 0) print(“ “)
   print(args(i))
   i += 1
}
println()
```

- Scala infiere el tipo `Int` de la variable `i` al asignarla el valor inicial `0`.
- La condiciones en `while` o `if` **siempre** entre paréntesis.
- Se puede usar ; al finalizar cada instrucción, no es obligatorio.
- El incremento `++i` o `i++` (utilizado en Java) no funciona en Scala. Usar `i = i + 1` ó `i += 1`

```
$ scala echoargs.scala Scala es muy divertido

Scala es muy divertido
```

Paso 6: Iteraciones

- `for` se usa para iterar sobre todos los elementos de una colección.

```
/* For.scala */
for(archivo <- archivos)
      println(archivo)
```

- La expresión entre paréntesis que aparece en el `for` se denomina ***generador***.
- La sintaxis para un generador es:

  `<identificador-val> <- <expresión-generadora>`

- Un generador puede aparecer únicamente dentro de una expresión for.
- El tipo de la variable ***archivo*** es `val` y no `var`. ¿Por qué?

Paso 6: Iteraciones (y2)

- `foreach` se usa para aplicar una función sobre los elementos de una colección.

```
/* foreach.scala */
args.foreach(arg => println(arg))
```

```
$ scala foreach.scala Scala es divertido

Scala
Es
divertido
```

- Scala infiere el tipo de `arg` como `String`, ya que le estamos pasando caracteres.
- Podríamos haber usado:

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

Parametrizar arrays con tipos

```
val cadenasSaludo = new Array[String](3)
cadenasSaludo(0) = “¡Hola”
cadenasSaludo(1) = “,”
cadenasSaludo(2) = “ mundo!”

for (i <- 0 to 2)
   print(cadenasSaludo(i))
```

= cadenaSaludo.update=(0,”¡Hola”)

= (0).to(2)

= cadenaSaludo.apply(i)

- ¡¡OJO!! La forma de acceder a los elementos del array es a través de ().
- ¿Por qué se define val el array “cadenaSaludo”?
- val nombresNum = Array(“cero”, “uno”, “dos”) es equivalente a val nombresNum = Array.apply(“cero”, “uno”, “dos”)

Listas

- Son las estructuras de datos por excelencia en lenguajes de programación funcionales.
- En Scala son colecciones de objetos cuya estructura no puede alterarse (si no, serían Array).
- La clase que implementa las listas es la clase **List**:

```
var ejemploLista = List(1, 2, 3)
```

- Las listas son estructuras que pueden estar vacías o no:
  - Una lista no vacía, está formada por un objeto denominado **head** y por otra lista denominada **tail**.
  - La lista vacía se representa con el literal **Nil**, que es un objeto singleton.
- El operador "::" (cons) se utiliza para definir una lista a partir del **head** y el **tail**.
- El operador ":::" sirve para concatenar listas.
- No existe como tal la operación *añadir* un elemento a una lista.

Listas (y1)

```
/* listas.scala */
val lista1 = List(1, 2)
val lista2 = List(3, 4)
val lista3 = lista1 ::: lista2

println(lista1 + " y " + lista2 + " son listas en Scala")
println("Al concatenar las listas, el resultado es ", lista3)
```

```
$ scala listas.scala

List(1, 2) y List(3, 4) son listas en Scala
Al concatenar las listas, el resultado es List(1, 2, 3, 4)
```

- El símbolo más utilizado es "::". Así obtendríamos el mismo resultado que antes:

val lista3 = 1 :: 2 :: 3 :: 4 :: Nil

Listas (y2)

Método	Resultado
List() o Nil	Lista vacía
List("Uno", "Dos", "Tres")	Crea una lista con estos tres valores
val lista1 = "¡hola" :: "mundo" :: "!" :: Nil	Crea la lista lista1 con tres valores
List("Uno", "Dos"):::List("Tres")	Concatena las dos listas
lista1(2)	Devuelve "!"
lista1.head	Devuelve "¡hola"
lista1.tail	Devuelve una lista con "mundo" y "!"
lista1.reverse	Devuelve la lista en orden inverso
lista1.length	Devuelve el tamaño de la lista (3)
lista1.last	Devuelve "!"
lista1.isEmpty	Si la lista es vacía. Devuelve false

Tuplas

- Las tuplas son secuencias ordenadas de objetos, similares a las listas en el sentido de que son inmutables.
- Las tuplas en Scala se utilizan para agrupar de manera ordenada objetos, que en general son de tipos diferentes.
- **El uso ideal de una tupla es cuando se requiere retornar múltiples objetos en la invocación de un método.**
- Para instanciar una tupla, simplemente se colocan los objetos que la conforman entre paréntesis, separados por comas:

```
var tupla1 = (1, “uno”, List(1))
```

- Para acceder a un elemento de la tupla se hace de la siguiente forma:

```
tupla1._1 //Primer elemento
tupla1._3
```

Tuplas (y1)

- El tipo de una tupla depende del número de argumentos y el tipo de cada uno de ellos.

```
var tupla1 = (1, “uno”)
var tupla4 = (‘s’, ‘a’, 30, List(1))
```

Tipo Tuple2[Int, String]

Tipo Tuple4[Char, Char, Int, List[Int]]

- Se pueden definir actualmente tuplas de hasta 22 elementos en Scala.

Conjuntos

- Scala provee operaciones de conjuntos a través de la clase **Set**.
- A diferencia de las clases vistas anteriormente (Lista, Array), Scala permite trabajar con conjuntos mutables o inmutables. Por defecto, Scala trabaja con conjuntos inmutables.

```
var conjunto = Set[Int]()
```

Se debe especificar el tipo de un conjunto vacío

```
var conjunto = Set(1, 3, 5, 7, 11)
```

Conjunto con elementos: el compilador infiere el tipo de datos

Si se quiere trabajar con conjuntos mutables …

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

- Las operaciones más comunes con conjuntos son + y +=.

Conjuntos (y1)

```
var trilogias = Set(“El Padrino, “El Señor de los Anillos”)
trilogias += “Millenium”
println(trilogias)
```

- ¿Conjunto Mutable?
- Se crea un nuevo conjunto “trilogias” con el valor “Millenium”

```
import scala.collection.mutable.Set
val trilogias = Set(“El Padrino, “El Señor de los Anillos”)
trilogias += “Millenium”
println(trilogias)
```

- Se añade el elemento “Millenium” al conjunto “trilogias”

Mapas

- Scala tiene soporte para mapas a través de dos implementaciones para la clase Map:
  - Mutable
  - Inmutable
- Los mapas son contenedores asociativos para los que se definen **parejas de llave-valor.**
- Las llaves en Scala pueden ser cualquier objeto, usualmente son cadenas.

```
val m = Map(
   “hola” –> List(‘h’, ‘o’, ‘l’, ‘a’),
   “hi” -> List(‘h’, ‘i’))

println(m(“hola”))
```

- “hi” → Llave
- -> → Flecha derecha
- List(‘h’, ‘i’) → Valor
- **Mapa inmutable**

Mapas (y1)

```
/* mapas.scala */
import scala.collection.mutable.Map
val m = Map[Int, String]()
m += 1 -> “uno”
m += ((4, “cuatro”))

println(m(1))
```

- Creación de un mapa vacío →Definir tipo

```
$ scala mapas.scala

"uno"
```

Mapas (y2)

Método	Resultado
+=	Asigna/Agrega una nueva pareja al mapa
contains	Verifica si una llave está presente en un mapa
-=	Elimina una pareja del mapa especificando la llave de la misma
keys	Obtiene una colección iterable con las llaves del mapa
values	Obtiene una colección iterable con los valores del mapa

Aprender a reconocer el estilo funcional

- Programación Funcional vs Programación Imperativa
  - Si el código contiene vars → Probablemente programación imperativa
  - Si el código no contiene vars (y sí vals) → Probablemente programación funcional

```
def printArgs(args: Array[String]): Unit = {
   var i = 0
   while (i < args.length) {
      println(args(i))
      i += 1
   }
}
```

¿Imperativo o Funcional?

Aprender a reconocer el estilo funcional (y1)

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

Aproximación Funcional 1

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

Aproximación Funcional 2

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

println(formatArgs(args))
```

Aproximación Funcional 3

Ten en cuenta en Scala …

- Para tus programas en Scala, primeramente utiliza:
  - vals para la definición de variables.
  - objetos inmutables.
  - métodos sin efectos colaterales

- Si no es posible y por razones justificadas, entonces utilizar:
  - vars para la definición de variables.
  - objetos mutables.
  - métodos con posibles efectos colaterales

Leer líneas de un fichero

```
/* contarCaracteres.scala */
import scala.io.Source
if (args.length > 0) {
   for (line <- Source.fromFile(args(0)).getLines)
      print(line.length + “ “ + line)
}
else
   Console.err.println(“Por favor introduce un nombre”)
```

```
$ scala contarCaracteres.scala contarCaracteres.scala

23 import scala.io.Source
1
23 if (args.length > 0) {
1
50 for (line <- Source.fromFile(args(0)).getLines)
36    print(line.length + “ “ + line)
2  }
5 else
53   Console.err.println(“Por favor introduce un nombre”)
```

Clases y objetos

Programación OO en Scala

- Scala es un lenguaje orientado a objetos
  - En el sentido de que todo es un objeto incluyendo números o funciones y también tienen métodos (1).+(((2).*(3))./(x))
  - Los tipos y comportamientos de objetos son descriptos por clases y traits
- Características de POO:
  - **Abstracción**: Objetos de software son abstracciones de objetos reales.
  - **Encapsulación**: Protección de datos.
  - **Polimorfismo:** Descripción de comportamiento común para objetos distintos.
  - **Herencia:** Relaciones de pertenencia que permiten la reutilización de código.

Programación OO: Clases

- Las clases son entidades de software que permiten la representación de conceptos dentro del lenguaje Scala.
- Su propósito principal es de servir de arquetipos o planos de esos conceptos.
- Las descripciones logradas a través de las clases permiten la organización y relación sistemática y estructurada de piezas de software: tanto datos como comportamiento de un programa.

Las Clases en Scala

- Las clases en Scala pueden **definirse** e **instanciarse**.

- En una **definición**, especificamos el nombre o identificador que queremos darle a la clase, su relación con otras clases y sus miembros.

- Al **instanciar una clase**, creamos un objeto o instancia particular de la misma. Podemos tener muchas instancias de una misma clase.

Definición de una clase en Scala

```
class Estudiante {
   val Nombre = “Juan”
   val Apellido = “Pérez”
   var Edad = 23
   var Estudios = “Grado en II”
}
```

Identificador de la clase

Campos

- Los **atributos se conocen como campos**. Son referencias a instancias de objetos que pueden definirse utilizando **var** o **val**.
- En POO los campos representan la **composición**, es decir, relación de pertenencia de un objeto a otro.

Métodos

```
class Estudiante {
   def obtenerEstudios():Unit = {
      println(Estudios)
   }
}
```

Para definir un método, se usa **def**

- Los métodos definen el comportamiento y se corresponden con la definición de funciones dentro de una clase.

Instancias de clase

```
new estudiante1 = new Estudiante
estudiante1.obtenerEstudios()
Estudiante1.obtenerEstudios()
```

- Una vez definida una clase, podemos **instanciarla** utilizando la palabra reservada **new** seguida por el nombre de la clase.
- Podemos crear varias instancias de una misma clase, cada instancia tiene su propio estado al ocupar regiones de memoria distintas
- A los campos de una clase también se les conoce como variables de instancia, pues cada instancia tiene su propia copia

Campos privados

```
class Estudiante {
   private var estado = “parado”
   def cambiarSituacionLaboral():Unit = {
      estado = “trabajando”
   }
}
```

Campo privado. Impide el acceso directo a “estado”

Accesible desde dentro de la clase

```
var estudiante1 = new Estudiante;

estudiante1.estado
```

¡¡ERROR!! Los campos **private** únicamente pueden accederse desde los métodos de la misma clase

Campos privados en clases internas

```
class Externa {
   var i = 1

   class Interna {
      private def f() = i + 1
   }
   def g() = (new Interna).f
}
```

OK

¡¡ERROR!! Los campos **private** tampoco pueden ser accedidos por otra clase, aunque esta clase defina a la primera

Campos públicos

```
class Estudiante {
   val Nombre = “Juan”
   val Apellido = “Pérez”
   var Edad = 23
   var Estudios = “Grado en II”
}
```

- Campos públicos.
- Definición en Scala por omisión.
- ¡OJO! En Java sí que habría que indicar **public**.

Parámetros de métodos

```
def f(x:Int):Unit = {
    x += 1
}
```

ERROR en tiempo de compilación

- Los parámetros de los métodos son siempre **val**
- **No se puede cambiar el valor de dicho parámetro.**

Clases y objetos en Scala

- **Class** para crear una clase y **new** para crear una instancia (objeto). Sintaxis:

```
class Clase {
   …
   definición de la clase
   …
}
```

```
new Clase
```

- Ejemplo:

```
class Ave {
   def volar() = println(“¡estoy volando!”)
}
```

```
var aguila = new Ave

aguila.volar
¡estoy volando!
```

Nota: todo es un objeto, incluyendo números o funciones y también tienen métodos (1).+(((2).*(3))./(x))

Campos y métodos en Scala

- Los **campos** (también llamados variables de instancia, porque cada instancia mantiene su propio conjunto de variables) mantienen el estado de los objetos. Se definen con **val** o **var**.
- Los **métodos** modifican el estado de los objetos. Se definen con **def**.

```
class Ave {
   var nombre = “aguila”
   def setNombre(n:String) =
      nombre = n
   def volar() =
      println(“¡estoy volando!”)
}
```

```
var miAguila = new Ave
miAguila.nombre --> String:ave
miAguila.setNombre(“Falcon”)
miAguila.nombre --> String:Falcon
```

```
def <nombre de la función>(Parametro1:tipo1>, …):<tipo_resultado>={
            <cuerpo de la función>}
```

Inferencia de punto y coma

```
var a=1; a += 1

var a=1
a += 1
```

El “;” se utiliza para separar dos instrucciones en una misma línea

Scala las trata como instrucciones diferentes, sin necesidad de utilizar “;”

Punto y coma

- Una nueva línea se trata como punto y coma a menos que:
  - **Termine en una palabra que no sería legal** como fin de instrucción.
  - La línea siguiente comienza con una **palabra que no sería legal** como inicio de instrucción.
  - La línea termina pero está dentro de “()” o “[]”.

Objetos Singleton

- Métodos y valores que no están asociados con instancias individuales de una clase se denominan objetos singleton y se denotan con la palabra reservada **object** en vez de class.
- Por definición corresponden con instancias únicas de una clase:
  - Si **a** es una instancia de la clase **A** y
  - **b** es una instancia de la clase **A**
  - Se cumple que **a** y **b** son el mismo objeto (ocupan la misma región de memoria)
- Se utilizan en Scala para reemplazar los miembros de clase estáticos que existen en Java.

Objetos Singleton: Definición

```
object O {
   var nombre = “Singleton de O”
   def salida() = println(nombre)
}

O.Out
```

Muy similar a la definición de una clase (**object** por **class**)

Única instancia

Singleton y clase acompañante

- Cuando se declara un objeto singleton con el mismo nombre que una clase, al objeto singleton se le conoce como **objeto acompañante** (*companion object*)
- La clase se llama **clase acompañante** del objeto singleton (*companion class*)
- Es necesario que la definición de la clase y de su objeto acompañante se haga dentro del mismo archivo.

Singleton standalone como aplicación

- Los objetos singleton pueden **no** tener una clase acompañante, en cuyo caso se les conoce como objetos singleton standalone.
- Un objeto singleton standalone se usa para encapsular y agrupar módulos de software, o como punto de entrada de una aplicación en Scala.

```
object miPrograma {
   def main(args:Array[String]) {
      println("Mi aplicación")
   }
}
```

Definir método **Main**

Operaciones y tipos básicos

Tipos básicos

- Los tamaños y rangos de los tipos básicos corresponden con aquellos del lenguaje Java.

Value type	Range
`Byte`	8-bit signed two's complement integer (-2^7 to $2^7 - 1$, inclusive)
`Short`	16-bit signed two's complement integer (-2^{15} to $2^{15} - 1$, inclusive)
`Int`	32-bit signed two's complement integer (-2^{31} to $2^{31} - 1$, inclusive)
`Long`	64-bit signed two's complement integer (-2^{63} to $2^{63} - 1$, inclusive)
`Char`	16-bit unsigned Unicode character (0 to $2^{16} - 1$, inclusive)
`String`	a sequence of `Chars`
`Float`	32-bit IEEE 754 single-precision float
`Double`	64-bit IEEE 754 double-precision float
`Boolean`	`true` or `false`

Literales constantes

- Constantes enteras. Se usan para establecer valores de tipos básicos como Int, Long y Short.
- Si el número comienza con 0x o 0X, se interpreta como una constante en base hexadecimal (0-9, A-F, a-f).
- Si el número comienzo con 0, se interpreta como una constante en base octal (0-7).

```
$ scala> val hex = 0x05
hex: Int = 5

$ scala> val oct = 035
oct: Int = 29
```

Literales constantes (y1)

- Si el número finaliza con la letra L o l, la constante se corresponde con el tipo Long.
- Literales de punto flotante:
  - Notación decimal (parte entera, punto, parte fraccional).
  - Notación científica (se usa E o e para indicar que el número está multiplicado por 10 elevado al número después de la letra e).

```
$ scala> val of = 31l  // val of = 31L
of: Long = 31
$ scala> val big = 1.2345
big: Double = 1.2345
$ scala> val littleBigger = 3e5f
littleBigger: Float = 300000.0
```

Literales de carácter

- Se encierran con comillas simples.
- Se puede especificar el código del carácter en base octal en el rango ‘\0’ a ‘\377’.
- Se puede especificar el código unicode del carácter en hexadecimal utilizando **u** después de ****.

```
$ scala> val a = ‘A’
a: Char = A

$ scala> val c = ‘\101’
c: Char = A

$ scala> val d = ‘\u0041’
d: Char = A
```

Caracteres especiales

Literal	Meaning
\n	line feed (\u000A)
\b	backspace (\u0008)
\t	tab (\u0009)
\f	form feed (\u000C)
\r	carriage return (\u000D)
\"	double quote (\u0022)
\'	single quote (\u0027)
\\	backslash (\u005C)

Literales de cadena

- Las cadenas se definen entre comillas dobles.
- El símbolo \\ permite “escapar” caracteres especiales.
- Se pueden utilizar triples comillas al principio y final de una cadena para indicar al compilador que no interprete la cadena.

```
$ scala> val hola = "hola"
hola: java.lang.String = hola
$ scala> val escapes = "\\\" \’"
escapes: java.lang.String = \"’
$ println("""Bienvenido a mi PC.
             Pulsa una tecla para continuar.""")
```

En pantalla se mostraría:
Bienvenido a mi PC.
Pulsa una tecla para continuar.

Literales booleanas

- Se utilizan las palabras reservadas true y false.

```
$ scala> val bool = true
bool: Boolean = true

$ scala> val fool = false
fool: Boolean = false
```

Operación y operadores

En matemáticas una operación, involucra uno o más operandos.

- El **número de operandos** se conoce como aridad de la operación.
- Un **operador** es uno o más símbolos que representan a la operación.

Ejemplo Operación SUMA: 2 + 3

→ Aridad 2, Operador ‘+’

Operadores

- En Scala los operadores representan métodos definidos en los tipos básicos.

Ejemplo:

```
(1).+(1)   //Equivalente a 1+1
```

Parámetro del método ".+"

Método "+"

Objeto tipo Int

Los números son objetos, estos también tienen métodos.

Una expresión aritmética como 1 + 2 * 3 / x

Consiste en llamadas a métodos, (1).+(((2).*(3))./(x))

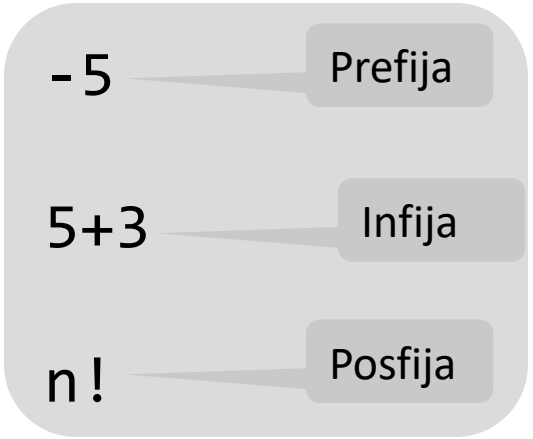
Sobrecarga de operadores

- Adicionalmente la clase **Int** implementa varios métodos para permitir la suma para tipos de argumentos que no son Int, el operador es el mismo, a esto se le denomina **sobrecarga**.

```
def + (arg0 : Int)    : Int

def + (arg0 : String) : String
def + (arg0 : Double) : Double
def + (arg0 : Float)  : Float
```

Notación de operadores

- En matemáticas se disponen de varias formas de escribir una operación, la sintaxis que relaciona a un operador con sus operandos puede ser:



Notación infija

- Scala soporta la notación infija en la que el operador se encuentra ubicado entre el primer y el resto de operandos, típicamente los operadores son binarios (por ejemplo, la suma), pero podría haber más de dos operandos, el **primer operando siempre es una instancia de la clase que define al operador**.

- **Todo** método definido en una clase puede ser invocado como un operador en notación infija.

Notación prefija

- Scala tiene un soporte muy restringido para notación prefija.
- Notación prefija es “unaria”, es decir, sólo tiene un operando.
- Sólo se pueden usar los símbolos “+”, “-”, “!” y “~”.
- Al definir el método es necesario anteponer **unary_** al símbolo.

```
$ scala> -2.0              //Scala llama (2.0).unary_-
res2: Double = -2.0

$ scala> (2.0).unary_-
res3: Double = -2.0
```

Utilizando el prefijo **unary_** para el operador, se indica a Scala que debe invertir el orden natural de invocación (notación infija)

Notación posfija

- Soporte de Scala para notación posfija es limitado. Sólo se puede utilizar con un operando (aridad 1).
- Se puede utilizar cualquier símbolo o identificador.
- Al definirlo, se define como cualquier otro método que no recibe argumentos.

```
$ scala> val s = "¡Hola Mundo!"
s: java.lang.String = ¡Hola Mundo!

$ scala> s.toLowerCase
res4: java.lang.String = ¡hola mundo!
```

Operadores y métodos

- Lo que convierte a un método en un operador, para Scala es la manera de utilizarlo (omitiendo el punto).

- Cualquier método puede usarse en notación infija y si no tiene argumentos en notación posfija.

- La convención establece que si un método tiene efectos secundarios (realiza algo adicional a retornar un resultado) se incluyan los paréntesis al invocarlo.

Operadores en tipos numéricos

- Para tipos básicos numéricos se tienen los operadores:

Operador	Operación
+	Suma
-	Resta
*	Multiplicación
/	División
%	Resto de la división entera

- Si se trata de tipos enteros, el operador “/” entrega la división entera.
- En tipos numéricos existe el operador “-” usado con notación prefija para invertir el signo (también existe “+”, sin efectos).

Operadores relacionales

- Se disponen de los siguientes operadores relacionales:

Operador	Operación
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que
==	Igual que
!=	Distinto de

- Todos estos operadores se usan en forma infija y retornan un valor de tipo Boolean.
- Las comparaciones con “==“ en Scala se hacen con el valor del objeto a diferencia de Java que es comparación de referencias.

Operadores lógicos

- Los operadores siguientes operan sobre operandos de tipo Boolean:

Operador	Operación
||, or	Disyunción
&&, and	Conjunción
!, not	Negación

- Se les denomina de “cortocircuito” porque cuando se combinan varios en una expresión lógica, los operandos se **evalúan de izquierda a derecha** hasta que se puede determinar su valor de verdad, de tal forma que operandos posteriores no son evaluados.

Operadores lógicos: Ejemplo

```
(1 > ‘a’) && (“hola”==“hola”)
```

El operando derecho sólo se evalúa si el izquierdo es verdadero.

```
false || true
```

El operando derecho sólo se evalúa si el izquierdo es falso.

Operadores de bits lógicos

- Los operadores siguientes operan sobre operandos de tipo Boolean:

Operador	Operación
& (bitwise-and)	Conjunción de cada bit
| (bitwise-or)	Disyunción de cada bit
~ (bitwise-not)	Negación de cada bit
^ (exclusive-or)	Disyunción exclusiva

- Operan sobre cada uno de los bits que representan el tipo numérico.
- El operador “^” es verdadero cuando sus operandos tienen valor de verdad distinto.

Operadores de bits lógicos: Ejemplo

```
var a = 8           // …1000
var b = 11          // …1011

a | b               // …1011
a & b               // …1000
~a                  // …0111
a ^ b               // …0011
```

Corrimiento en bits

- Scala adicionalmente permite el corrimiento de bits sobre la representación binaria de números enteros (Int).

Operador	Operación
>>	Corrimiento hacia la derecha (o hacia la parte menos significativa)
<<	Corrimiento hacia la izquierda (o hacia la parte más significativa)
>>>	Corrimiento sin signo derecho. También recorre el bit de signo (bit más significativo)

Corrimiento en bits: Ejemplo

```
var a = 0xF          // 00…01111

a >> 1               // 00…00111
a << 1               // 00…11111

var b = -1           // 11…11111

b >> 1               // 11…11111
b << 1               // 11…11110
b >>> 1              // 01…11111
```

Precedencia de operadores

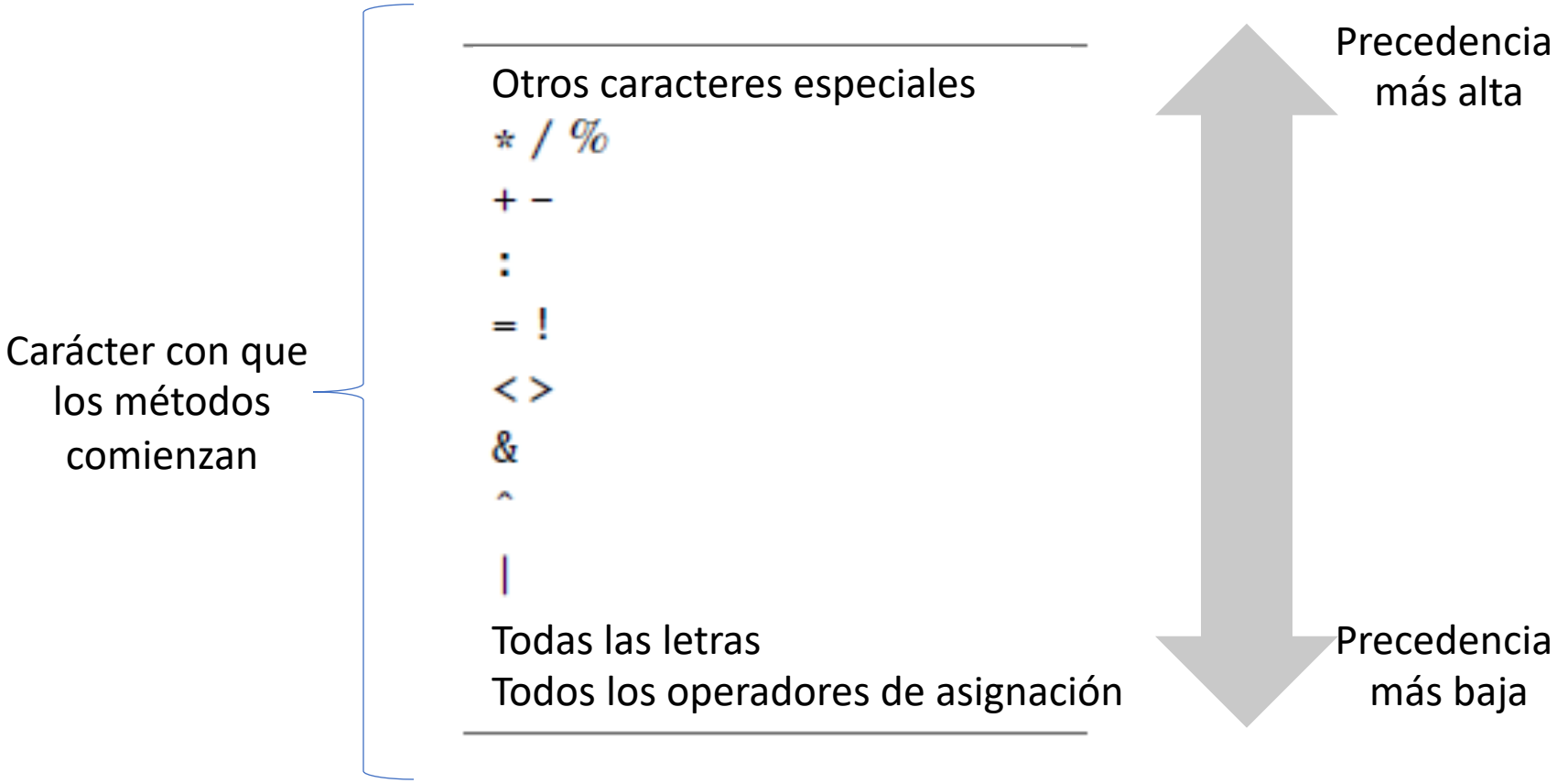
- Dentro de una expresión, los operadores se evalúan de acuerdo a su posición en la misma y la precedencia del operador.

```
1+3*5

Resultado = 16
```

El resultado es 16 y no 20 porque el operador * tiene mayor precedencia que el operador +

- Dado que Scala no tiene operadores como tales, sino más bien notación de operadores sobre métodos de una clase, la precedencia se determina por el primer carácter del método.

Precedencia



Precedencia (y1)

- Los operadores de asignación, es decir, aquellos métodos cuyo nombre termina en el símbolo “=“ y no son alguno de los operadores de comparación (<=, >=, ==, !=) tienen la precedencia más baja de todas.
- Si dos operadores tienen la misma precedencia, la asociatividad del operador determina cómo se agrupan las operaciones.
- La asociativad se determina por el último carácter en el nombre de un método.

Precedencia (y2)

```
x *= y + 1

Es idéntico a

x *= (y + 1)
```

*= es considerado como operador de asignación y la precedencia es menor que **+**, aunque se pueda pensar que se mayor por el símbolo ***

Asociatividad

- Si el nombre de un método termina con el símbolo “**:**”, la asociatividad es de derecha a izquierda, siendo de izquierda a derecha en cualquier otro caso.

```
a++b  //lo mismo que a.++(b)

a+:b  //lo mismo que b.+:(a)
```

Clases Rich Wrapper

- Para cada uno de los tipos básicos existe una clase Wrapper que provee un conjunto extendido de operadores (métodos).
- Se tiene acceso a estos Wrappers de forma automática a través de conversiones implícitas entre el tipo básico y el tipo Wrapper.
- La documentación de cada una de estas clases se puede consultar en:

  http://www.scala-lang.org/api/current/index.html

Clases Rich Wrapper (y1)

Basic type	Rich wrapper
Byte	scala.runtime.RichByte
Short	scala.runtime.RichShort
Int	scala.runtime.RichInt
Char	scala.runtime.RichChar
String	scala.runtime.RichString
Float	scala.runtime.RichFloat
Double	scala.runtime.RichDouble
Boolean	scala.runtime.RichBoolean

Clases Rich Wrapper: Ejemplos

Código	Resultado
0 max 5	5
0 min 5	0
-2.7 abs	2.7
-2.7 round	-3L
1.5 isInfinity	false
(1.0/0) isInfinity	true
4 to 6	Range(4, 5, 6)
"bob" capitalize	"Bob"
"robert" drop 2	"bert"

Objetos funcionales

Objetos funcionales

- Los objetos funcionales son aquellos que no cambian su estado, es decir, son inmutables.

- El que no puedan cambiar su estado, no significa que no tengan estado, para crear el estado del objeto, se utilizan parámetros de clase.

- Los parámetros de clase para objetos funcionales deben ser de tipo val (tipo de parámetro por omisión)

Objetos funcionales: Ejemplo

```
class Complejo(r:Double,
               i:Double)
```

Un número complejo no cambia su valor, pueden aplicarse operadores a uno o varios números complejos, el resultado es un nuevo número complejo

Dos parámetros de clase tipo **val**. Scala creará un constructor principal con los dos parámetros

Scala vs Java

- Las clases en Scala pueden especificar parámetros. Esto es diferente de Java donde los métodos constructores pueden especificar parámetros que requieren de copiarse a miembros privados de la clase.

- La aproximación de Scala es más concisa, limpia y menos propensa a errores, especialmente en clases pequeñas.

Constructores en Java y Scala

```
public class Person {
        public Person(int age) {
        if (age > 21) {
            System.out.println("Over drinking age in the US");
        }
        }

        public void dance() { ... }
    }

    Person p = new Person(26);   // prints notice that person is overage
```

```
class Person(age:Int) {
      if (age > 21) {
        println("Over drinking age in the US")
      }

      def dance() = { ... }
    }

    var p = new Person(26)   // prints notice that person is overage
```

Constructor principal

- El compilador de Scala ubicará todo el código que esté dentro del cuerpo de definición de una clase, y que no forme parte de un campo o método, dentro del ***constructor principal***.

```
class Complejo(r:Double,
               i:Double) {
   println(“[“ +r+ “+” +i+ “i]”)
}
```

```
var c = new Complejo(1,1)
```

Sobreescribiendo toString

- Es mejor cambiar el método **toString**:

```
class Complejo(r:Double,
               i:Double) {
   override def toString() = {
      “[“ + r + (if(i<0) “” else
       “+”) + i + “i]”
   }
}
```

Se antepone **override** a la definición de un método para indicarle al compilador que deseamos cambiar el comportamiento de algún método de una clase padre.

Parámetros de clase

- Si no se indica otra cosa, los **parámetros de clase**, además de ser de **tipo val**, sólo pueden **accederse en la instancia de clase** que los crea, es decir, tienen un comportamiento diferente a los campos.
- Un campo privado puede accederse desde cualquier instancia de la misma clase.
- **Para permitir el acceso a parámetros de clase**, podemos **convertirlos en campos** o definir métodos de acceso.

Parámetros de clase: Ejemplo

```
class Complejo(r:Double,
               i:Double) {
   val real = r
   val imaginaria = i
   override def toString() = {
      “[“ + real + (if(i<0) “” else
       “+”) + imaginaria + “i]”
   }
}
```

Creamos campos donde copiamos los parámetros de la clase, consiguiendo así que sean accesibles a otros objetos.

Implementando la suma

```
def + (c:Complejo) {
    new Complejo(real + c.real,
       imaginario + c.imaginario)
}
```

En esta nueva definición los campos son accesibles a través de esta y otras instancias de la clase **Complejo**, al ser públicos.

Campos paramétricos

```
class Complejo(val r:Double,
               val i:Double) {
   override def toString() =
      “[“ + r “+” + i + “i]”

   def +(C:Complejo) =
      new Complejo(r + c.r, i + c.i)
}
```

Una versión más compacta de la clase al usar los **campos paramétricos**.

Los namespaces de Java son: campos, métodos, tipos y paquetes

Los namespaces de Scala son: valores (campos, métodos, paquetes y objetos singleton) y tipos (clases y *traits*)

La razón de que Scala sólo tenga 2, es precisamente para que los métodos sin parámetro se puedan sobreescribir con un val.

Referencia propia: this

- La palabra reservada ***this*** se utiliza como una referencia de una instancia de una clase a sí misma.
- Puede utilizarse en métodos de la clase.

```
def +(c:Complejo) {
   new Complejo(this.r + c.r,
                this.i + c.i)
}
```

En este ejemplo, el uso de **this** es innecesario.

Constructores auxiliares

- En Scala se pueden tener varios métodos constructores (varias formas de crear una instancia de clase) estos métodos se denominan ***constructores auxiliares***.
- El nombre de los métodos auxiliares tiene que ser ***this***, pudiendo especificar el número y tipo de parámetros que se desee.
- La primera línea en todo constructor auxiliar debe invocar otro constructor de la misma clase, ya sea primario o auxiliar.

https://www.oreilly.com/library/view/scala-cookbook/9781449340292/ch04s04.html

Constructor auxiliar: Ejemplo

- Podemos crear un constructor auxiliar para permitir crear números complejos a partir de un número real.

```
def this(real:Double) =
    this(real, 0)
```

Invocación al constructor principal

this indica que es un constructor auxiliar

Sobrecarga operador

- Podemos sobrecargar el operador “+” para que funcione con números reales.

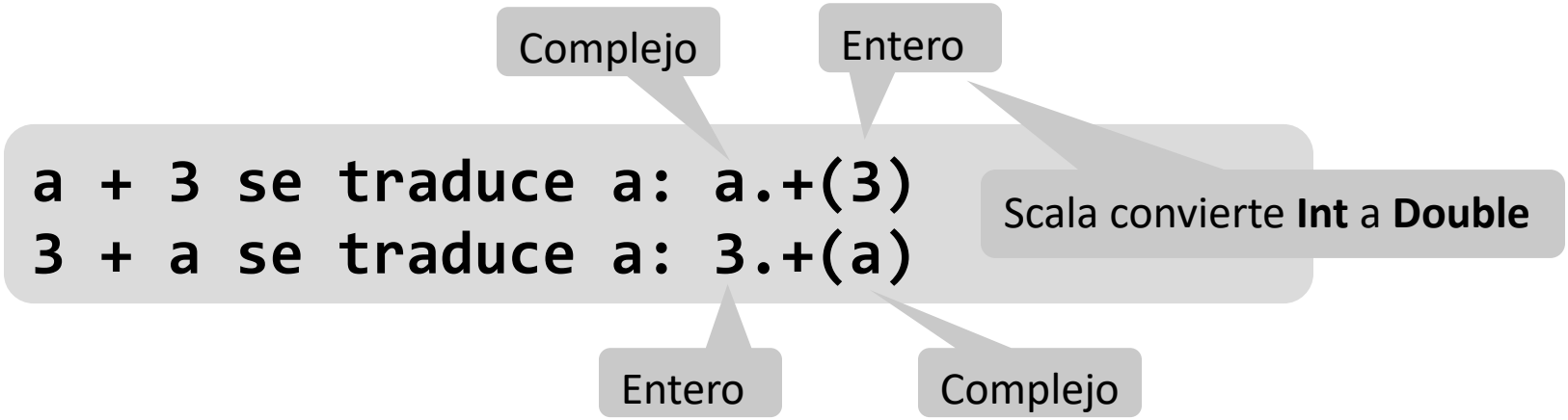
```
def +(r:Double) =
    new Complejo(this.r+r, i)
```

El uso de **this** permite distinguir el campo del parámetro del método

```
var a = new Complex(1, 1)
a + 3 // Podemos sumar reales
3 + a // ☹
```

Conversiones implícitas

- En el ejemplo anterior …



La clase Int no sabe cómo sumar complejos, para resolver esto, podemos utilizar ***conversiones implícitas***

Conversiones implícitas (y1)

- Al definir la conversión implícita, Scala hace la conversión para poder realizar la operación:

```
implicit def doubleAComplejo(x:Double) =
   new Complejo(x)
```

- Para que la conversión pueda usarse se requiere definir ***doubleAComplejo*** en un módulo u objeto acompañante.

Estructuras de control

Estructuras de control

- Scala tiene un conjunto reducido de estructuras de control:
    - if
    - while
    - for
    - try – catch
    - match

Condiciones: if

- La estructura de control **if**, permite decidir sobre la ejecución de dos secuencias de instrucciones de manera condicional sobre una expresión que evalúa a verdadero o falso.

- Funciona de manera análoga al operador ternario del lenguaje **C**, es decir un bloque **if** retorna un valor al completar su ejecución.

Condiciones: if - else

```
if (<expresión>) {
   <instrucciones si true>
   <valor retorno del bloque>
} else {
   <instrucciones si false>
   <valor retorno del bloque>
}
```

```
var nombreFichero = "default.txt"
if (!args.isEmpty)
   nombreFichero = args(0)
```

match

- La sentencia match permite evaluar una variable o expresión y comparar el resultado con un conjunto de opciones.
- Al igual que la sentencia if, la sentencia devuelve el último valor que evalúa

```
def pruebaMatch(str: String) = {
      str match {
case "salt" => println("pepper")
case "chips" => println("salsa")
case "eggs" => println("bacon")
case _ => println("huh?")
      }
 }
```

```
def pruebaMatch2(str: String): String = {
      str match {
case "salt" => "pepper"
case "chips" => "salsa"
case "eggs" => "bacon"
case _ => "huh?"
      }
}
```

```
pruebaMatch("eggs")          //> bacon
pruebaMatch2("eggs")         //> res11: String = bacon
```

Iteraciones: while

- La estructura de control **while** funciona de la misma forma que en otros lenguajes de programación: sirve para repetir una secuencia de instrucciones mientras cierta condición se cumpla, la ejecución de bloque no se da (o repite) cuando la condición se hace falsa.
- While siempre retorna el valor Unit().
- Permite la programación imperativa.
- **En Scala, se trata de evitar su utilización.**

Ejemplo: while

```
var i = 0
var A = new Array[Int](10)
while (i < 10) {
   println(“Iteración: “ + i)
   A(i) = i + 1
   i+=1
}
```

Iteraciones: do - while

- El bloque **do-while** sirve para ejecutar una secuencia de instrucciones y repetir la ejecución del bloque hasta que una condición se haga falsa.

- Como en el bloque **while**, el valor de retorno del bloque **do-while** es **Unit**.

- **Debe evitarse su uso en la medida de lo posible.**

Ejemplo: do-while

```
var i = 0
var colores = Array(“rojo”, “blanco”, “azul”,
“amarillo”)
do{
   println(“Mis colores preferidos son: “ +
colores(i))
   i+=1
} while(i<colores.length)
```

Iteraciones: Bucles

- Las estructuras de control para iteraciones **while** y **do-while** tienen el propósito de permitir programación imperativa, pero en lenguajes puramente funcionales no se cuenta con ellas.

- En lenguajes funcionales los bucles son convertidos a invocaciones recursivas sobre una misma función.

Iteraciones – Bucles: Ejemplos

```
def mcd(n:Long, m:Long): Long = {
   var a = n; var b = m
   while (a != 0) {
      val aux = a
      a = b % a
      b = aux
   }; b
}
```

ESTILO IMPERATIVO

```
def mcd(n:Long, m:Long): Long = {
   if (m == 0) n
   else mcd(m, n % m)
}
```

ESTILO FUNCIONAL

Iteraciones: for

- Martín Odersky nombra a las expresiones **for** una “navaja suiza” de las iteraciones.
- Puede usarse de manera muy simple para iterar sobre todos los elementos de una colección.

```
import java.io.File
val archivos = new File(“.”).listFiles
for (archivo <- archivos)
   println(archivo)
```

Sintaxis for

- En el ejemplo anterior la expresión entre paréntesis que aparece en la instrucción **for** se denomina **generador**.
- Sintaxis para un generador:

```
<identificador-val>
<-
<expresión-generadora>
```

- Un generador puede aparecer únicamente dentro de una expresión **for**.

```
for( a <- 1 to 3; b <- 1 to 3){
println( "Value of a: " + a ); println( "Value of b: " + b );
```

Expresión for – método foreach

- Iterar con un ciclo **for**, utilizando una expresión generadora como la del ejemplo anterior es posible porque las clases utilizadas como generador, todas implementan un método llamado **foreach** con la firma adecuada.

- Sin embargo las expresiones for, pueden ser mucho más flexibles, permitiendo por ejemplo, filtrar valores o retornar un valor.

```
val x = List(1,2,3)
 x: List[Int] = List(1, 2, 3)
scala> x.foreach { println }
```

Filtros – for

- Los filtros representan selectividad sobre los valores de una colección.
- Para especificar que deseamos iterar sobre sólo algunos elementos de la colección, se expresa la condición de filtrado tras la expresión generadora.

```
for (i<-1 to 10 if (i%5!=0))
   println(i)
```

El filtro se especifica dentro de las cláusulas que aparecen dentro de los paréntesis del **for**

Refinamiento de filtros

- Se puede filtrar sobre el resultado de un filtro; para expresar esto, basta con hacer seguir la condición **if** del segundo filtro a la condición del primero.

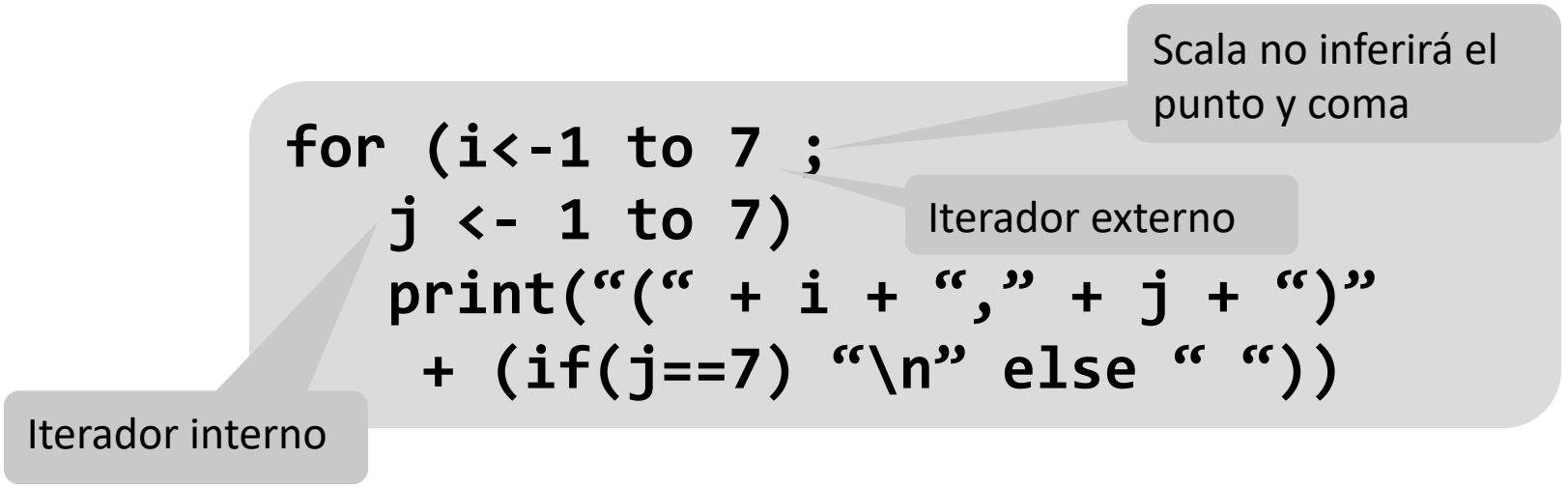
```
for (i<-1 to 10
   if (i%7!=0)
   if (i>5)
) println(i)
```

Primer filtro

Podemos concatenar filtros simplemente escribiendo la secuencia de las condiciones que los definen

Generadores múltiples

- Se pueden tener generadores múltiples para una misma expresión **for**.
- El resultado corresponde con el de **anidar** un iterador dentro del otro



Generadores múltiples

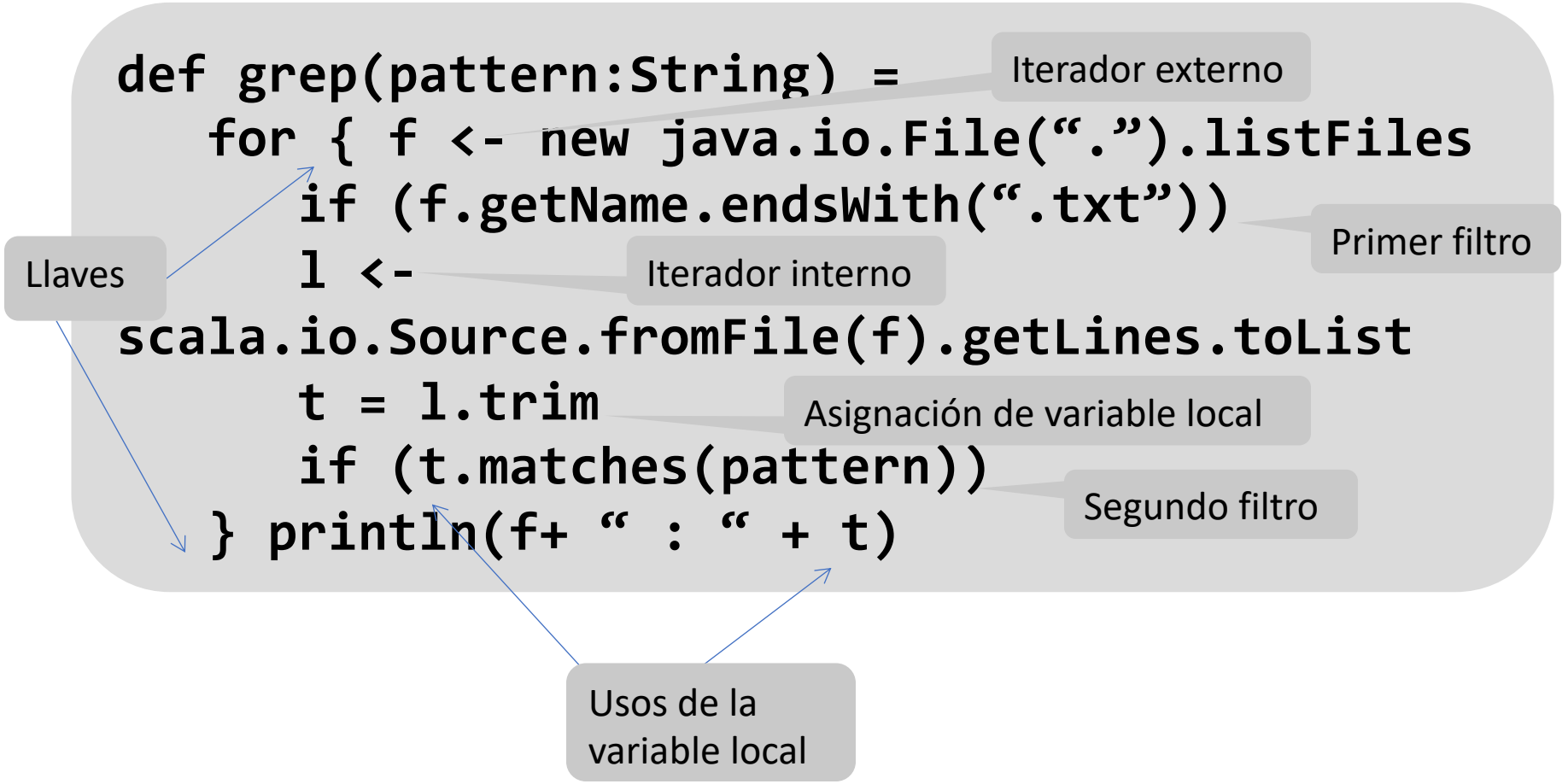
```
var a = Array(Array(1,2,3),
   Array(4,5,6), Array(7,8,9))
var b = new Array[Array[Int]](3)

for (i<-0 until 3)
   b(i) = new Array[Int](3)

for (i<-0 to 2; c=a(i); j<-0 to 2)
   b(j)(i)=c(j)
```

c es una nueva variable definida dentro de las cláusulas de la expresión for

Asociación de variables - for

- La asociación de variables puede lograrse introduciendo la misma con el símbolo “=“
- Estas asociaciones de variables deben aparecer tras una expresión generadora.
- Las variables así asociadas, corresponden con variables tipo **val**, y siempre refieren a variables locales nuevas, el nombre de la variable “opacará” cualquier nombre idéntico definido en otro ámbito externo.

Sintaxis – Usando {}



Expresiones for - yield

- En Scala las expresiones **for** pueden entregar un valor como resultado.
- El valor es una colección, los elementos de la cual se van agregando al utilizar la palabra reservada **yield**.
- La sintaxis es:

```
for <cláusulas> yield <cuerpo>
```

Debe aparecer entre las cláusulas de la expresión for y su cuerpo

for (i <- 1 to 5) yield i * 2

Ejemplo: yield

```
def normaliza(a:List[Double]) = {
   def max(a:List[Double]):Double =
      a match{
         case Nil    => Double.NaN
         case x::Nil => x
         case x::y   => x max max(y)
   }; val M = max(a)
   for (e <- a) yield e/M
```

try-catch

- Un método o función puede terminar de manera inesperada al existir una condición de error o excepción.
- En lugar de retornar un valor, el método o función “lanza” una excepción, que puede o no capturarse en otra parte del código.
- La forma de lanzar una excepción es la misma que en Java, por ejemplo:

```
throw new IllegalArgumentException
```

try es de tipo Nothing

- Nothing es un tipo especial en Scala que es subtipo de todos los demás tipos:

```
def div(x:Int, y:Int):Int =
   if (y != 0) x / y
else
   error(“división entre cero”)
```

- La función div, no viola la definición de tipos pues Nothing es subclase de Int

Atrapando excepciones - catch

- Se pueden atrapar excepciones, siempre que el código que puede generarlas se contenga dentro de un bloque **try**.

- Se atrapan las excepciones siempre que estas correspondan con las especificadas para los **case** dentro del bloque **catch**, si la excepción no aparece en ninguno de los tipos enunciados, la excepción sigue propagándose.

Sintaxis try-catch

```
try {
   <código con excepción potencial>
} catch {
   case <id-1>:<Tipo-1> =>
      <manejador excepción 1>
   case <id-2>:<Tipo-2> =>
      <manejador excepción 2>
}
```

```
try {
   error(“error”)
}catch{
   case r:RuntimeException => {
      println(“manejo de error”)
   }
}
```

Expresión try-catch-finally

- La estructura de control try-catch-finally puede usarse también para retornar un valor:

```
val divByZero = try {
   1/0
} catch {
   case e:ArithmeticException =>
Double.PositiveInfinity
} //java.lang.ArithmeticException
```

Sintaxis try-catch-finally

```
Try {
   //código principal
} catch {
   //manejo de excepciones
} finally {
   //código garantizado
}
```

Dentro del bloque **finally** se garantiza la ejecución de las instrucciones, se genere excepción o no dentro del bloque principal

Precaución finally

- Hay que tener cuidado al retornar valores de un bloque finally

```
def f(): String =
   try {
      return “try”
   } finally {
      return “finally”
   }
```

```
def f(): String =
   try {
      “try”
   } finally {
      “finally”
   }
```

Es buena práctica evitar retornar valores en el bloque **finally**

Composición y herencia

Clases y objetos en Scala

- **Herencia** es cuando un objeto o clase se basa en otro objeto o clase, usando la misma implementación o comportamiento.
  - reutilización de código mediante clases públicas e interfaces.
  - polimorfismo, donde hijas pueden adoptar la forma padre
  - herencia me permite sobrescribir métodos,
  - herencia se relaciona en una forma 1:1.

- **Composición** quiere decir que tenemos una instancia de una clase que contiene instancias de otras clases que implementan las funciones deseadas.
  - La composición por si misma no es polimórfica.
  - composición podemos elegir si vamos a tener 0, 1 o N

Clases y objetos en Scala

- ¿Qué pasa si defino la variable que contiene un objeto como val? ¿Podré modificar el objeto al que referencia? ¿Por qué?

```
val otroAguila = new Ave

otroAguila.setNombre(“Calva”)
```

Clases y objetos en Scala

- Todas las clases en Scala heredan de una super-clase
- En Scala es posible sobreescribir métodos heredados de una superclase.
- Pero es obligatorio especificar explícitamente que un método sobreescribe a otro usando el modificador **override**, para evitar sobreescrituras accidentales.
- Como **ejemplo** podemos aumentar nuestra clase Complex con la redefinición del método toString heredado de Object.

```
class Complex(real: double, imaginary: double) {
def re = real
def im = imaginary
override def toString() = "" + re + (if (im < 0) "" else "+") + im + "i”
}
```

Herencia y sobreescritura

- Para instanciar una clase es necesario usar la primitiva **new,** como se muestra en el siguiente ejemplo:

```
object Classes {
  def main(args: Array[String]) {
    val pt = new Point(1, 2)
    println(pt)
    pt.move(10, 10)
    println(pt)
  }
}
```

El programa define una aplicación ejecutable a través del método **main del objeto singleton Classes.**

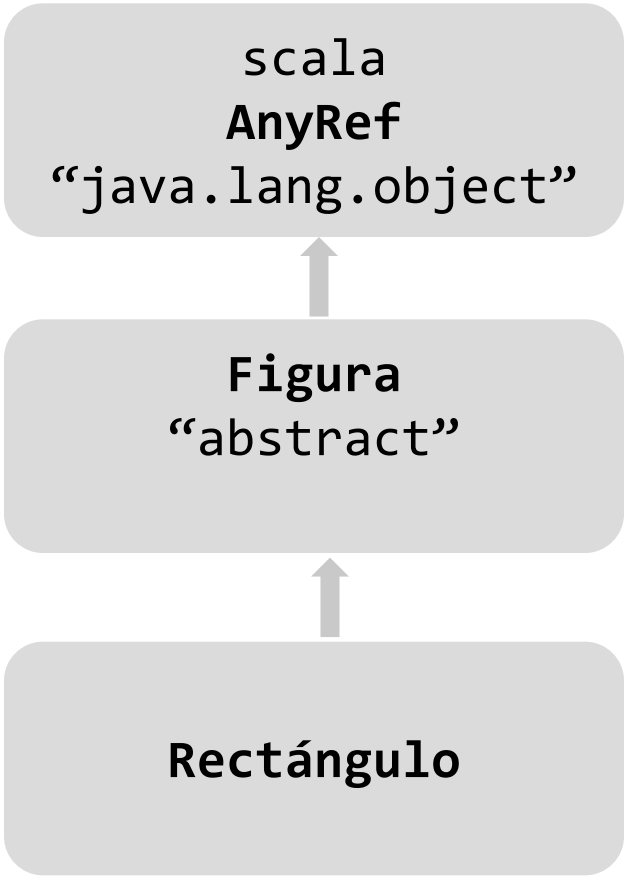
El método main crea un nuevo Point y lo almacena en pt.

Extendiendo clases

```
subClase extends superClase
```

- La cláusula extends produce dos efectos:
  - Hace que la subclase **herede** todos los miembros (campos y métodos) no privados de la superclase.
  - Hace que la subclase sea de un **subtipo** de la superclase.
- Si no se pone extends, el compilador implícitamente asume que la clase extiende de **scala.AnyRef** (lo mismo que **java.lang.object**).

Extendiendo clases (Jerarquía)



Objetos singleton para crear aplicaciones. Ejemplo

```
abstract class Figura {
   var nombre: String
   var coordSupIzq : (Int, Int)
   var coordInfDer : (Int, Int)
   def print(): Unit
   def setCoords(c1:(Int,Int),
                 c2:(Int,Int)) = {
      coordSupIzq = c1
      coordInfDer = c2
   }
}

class Rectangulo(name: String)
      extends Figura {
   var nombre = name
   var coordSupIzq = (0, 0)
   var coordInfDer = (0, 0)
   def print(): Unit = {
      println("Coord 1: " + coordSupIzq)
      println("Coord 2: " + coordInfDer)
   }
}
```

```
object Formas {
   def main (args: Array[String])={
      val rec = new Rectangulo("rec1")
      rec.setCoords((5,5),(98,77))
      rec.print
   }
}
```

```
> scalac Formas.scala
> Scala Formas
Coord 1: (5,5)
Coord 2: (98,77)
```

Creamos el fichero Formas.scala. Este nombre del fichero debe ser el mismo que el nombre de la aplicación → del objeto singleton

Extendiendo clases (herencia). Ejemplo

```
class Rectangulo(name: String) extends Figura {
   var nombre = name
   var coordSupIzq = (0, 0)
   var coordInfDer = (0, 0)
   def print(): Unit = {
      println("Coord 1: " + coordSupIzq)
      println("Coord 2: " + coordInfDer)
   }
}
```

```
var r1 = new Rectangulo("rec")
r1: Rectangulo = Rectangulo@dd841

r1.setCoords((1,1),(20,20))

r1.print
Coord 1: (1,1)
Coord 2: (20,20)
```

Método implementado en la superclase Figura

Clases abstractas

- Una **clase abstracta** puede tener alguno de sus miembros sin implementar.
  - es una clase que no se puede instanciar.
- Esto es coherente dado que una clase abstracta no tiene completa su implementación y encaja bien con la idea de que algo abstracto no puede materializarse.

```
abstract class Figura {
   var nombre: String
   var coordSupIzq : (Int, Int)
   var coordInfDer : (Int, Int)
   def print(): Unit
   def setCoords(c1:(Int, Int), c2:(Int, Int)) = {
      coordSupIzq = c1
      coordInfDer = c2
   }
}
```

Se definen colocando el modificador **abstract** delante de class.

Métodos sin parámetros

- Ya hemos visto que en Scala se puede llamar a los métodos que no tienen parámetros de un objeto sin necesidad de los paréntesis de argumentos:

```
aguila.volar
¡estoy volando!
```

- A la hora de definir estos métodos también es posible hacerlo sin utilizar parántesis de los argumentos:

```
abstract class Figura {
   …
   def print(): Unit
}
```

def print: Unit

clases Case. Ejemplo

```
abstract class Term
case class Var(name: String) extends Term
case class Fun(arg: String, body: Term) extends Term
case class App(f: Term, v: Term) extends Term
}
Fun("x", Fun("y", App(Var("x"), Var("y"))))
```

- la construcción de instancias de clases Case, no requiere que se utilice la primitiva new.
- Solo tiene sentido definir una clase Case si el reconocimiento de patrones es usado para descomponer la estructura de los datos de la clase

```
object TermTest extends scala.App {
  def printTerm(term: Term) {
    term match {
      case Var(n) =>
        print(n)
      case Fun(x, b) =>
        print("^" + x + ".")
        printTerm(b)
      case App(f, v) =>
        print("(")
        printTerm(f)
        print(" ")
        printTerm(v)
        print(")")
    }
  }
```

```
def isIdentityFun(term: Term): Boolean =
term match {
    case Fun(x, Var(y)) if x == y => true
    case _ => false
  }
  val id = Fun("x", Var("x"))
  val t = Fun("x", Fun("y", App(Var("x"),
Var("y"))))
  printTerm(t)
  println
  println(isIdentityFun(id))
  println(isIdentityFun(t))
}
```

Namespaces

- Scala tiene dos ***namespaces*** mientras que Java tiene cuatro:

JAVA
Campos
Métodos
Paquetes
Tipos

SCALA
Valores (campos, métodos, paquetes y objetos singleton)
Tipos (clases y traits)

- La razón de que Scala sólo tenga dos, es precisamente para que los métodos sin parámetro se puedan sobreescribir con un **val**.

Overriding de métodos y campos

- En Scala los métodos y los campos se encuentran en el mismo ***namespace***. Esto hace que sea posible que un campo sobreescriba un método sin parámetros:

```
abstract class Persona
   def misAmigos():
      ListBuffer[String]
}
```

```
class Mago(friends:ListBuffer[String]) extends
Persona{
   val misAmigos:ListBuffer[String] = friends
}
```

- Por este mismo motivo, en Scala no está permitido definir un campo y un método con el mismo nombre en la misma clase (en Java sí que se puede)

Definición de campos paramétricos

- Si volvemos a la definición de la clase **Rectangulo**, observamos que tiene un parámetro (name) cuyo único propósito es ser copiado al campo ***nombre***:

```
class Rectangulo(name: String) extends Figura {
   var nombre = name
   …
}
```

- Podemos ahorrar código si combinamos el parámetro y el campo definiendo un campo paramétrico:

```
class Rectangulo(val nombre: String) extends Figura {
…
}
```

Estamos definiendo al mismo tiempo un parámetro y un campo con el mismo nombre

Modificadores

- **override**: es necesario en aquellos miembros que sobreescriben un miembro concreto de la superclase.
  - Es opcional si la superclase es abstracta y no tiene implementado ese miembro.
- **private**: funcionan como en Java, sólo los miembros privados son visibles dentro de la clase u objeto que contienen la definición del miembro. Por defecto, todos los miembros son públicos.

```
class Gato {
   val peligroso = false
}
```

También se pueden añadir modificadores a los campos paramétricos

```
class Tigre(
   override val peligroso: Boolean,
   private var edad: Int
) extends Gato
```

Modificador final

- Funciona como en Java, cuando en una jerarquía de herencia, queremos asegurarnos que una subclase no pueda sobreescribir un miembro determinado. Se añade el modificador ***final*** delante del miembro.

```
abstract class Figura {
   var nombre: String
   var coordSupIzq : (Int, Int)
   var coordInfDer : (Int, Int)
   def print(): Unit
   final def setCoords(c1:(Int,Int), c2:(Int,Int)) = {
      coordSupIzq = c1
      coordInfDer = c2
   }
}
```

Si no queremos que la subclase **Rectangulo** sobreescriba el método **setCoords** de **Figura**

Object

- La palabra **object** define un objeto ***singleton***, una única instancia de una clase dada. Por sí mismo, no define un tipo.
- Los objetos ***singleton*** no pueden tener argumentos ya que para crearlos no se llama a **new** (la propia definición de **object** lo crea).

```
object FiguraMath {
   val pi = 3.14159
   def cuadrado(x: Int) = x * x
   def abs(x: Int) = if (x < 0) x * (-1)
}
```

Se define igual que una clase, pero en lugar de usar **class** se usa **object**.

```
FiguraMath.abs(-4)
def area (c:Circulo) = FiguraMath.cuadrado(c.radio) * (2 *
FiguraMath.pi)
```

Objetos singleton para crear aplicaciones

- Hasta el momento, en Scala hemos estado utilizando el intérprete o creando scripts (que no dejan de ser expresiones evaluadas).

- Para ejecutar un programa en Scala, se debe definir un objeto ***singleton*** que contenga el método **main**.

- El método main toma un **Array[String]** como argumento y es de tipo **Unit**.

Objetos singleton para crear aplicaciones

Cómo crear una aplicación

Al no tener métodos estáticos la forma de hacer un "main" es con objetos, como sigue:

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

Una variante a esto es crear un objeto que extienda de App

```
object Pepital extends App {

    println("hola pepita")

}
```

Objetos singleton para crear aplicaciones. Ejemplo

```
object Temporizador {
def unaVezPorSegundo(callback: () => Unit) {
while (true) { callback(); Thread sleep 1000
}
}
def tiempoVuela() {
println("El tiemplo vuela como una
flecha...")
}
def main(args: Array[String]) {
unaVezPorSegundo(tiempoVuela)
}
}
```

Salida compilando desde un archivo .scala

```
Welcome to the Scala worksheet
El tiemplo vuela como una flecha...
El tiemplo vuela como una flecha...
El tiemplo vuela como una flecha...
El tiemplo vuela como una flecha...
```

Traits

Trait Application

- Scala proporciona un ***trait*** llamado **scala.Application**. Este ***trait*** tiene declarado el método **main**, por lo que si lo extiende un objeto **singleton**, se puede usar como una aplicación de Scala directamente.

```
object Formas extends Application {
   val rec = new Rectangulo(“rec1”)
   rec.setCoords((5,5),(98,77))
   rec.print
}
```

```
> scalac Formas.scala
> Scala Formas
Coord 1: (5,5)
Coord 2: (98,77)
```

Traits

- Un ***trait*** encapsula definiciones de métodos y campos, que se pueden reutilizar mezclándolos dentro de clases.
- A diferencia de la herencia, donde una clase sólo puede heredar de una superclase, una clase puede **mezclar cualquier número de traits**.
- Se podría considerar que un trait es como una interfaz en Java con métodos concretos. Pero además los ***traits*** pueden declarar campos y mantener estado. Un ***trait*** define un nuevo tipo. Se consideran como **interfaces de Java enriquecidas**.

Traits (y1)

```
trait Rectangular {
   def coordSupIzq: (Int, Int)
   def coordInfDer: (Int, Int)
   def izq = coordSupIzq._1
   def der = coordInfDer._2
   def ancho = der – izq
}
```

Se define igual que una clase, pero en lugar de usar **class** se usa **trait**.

- Para que una clase utilice (mezcle) un trait, se pueden poner dos formas:
  - Con **extends**, significa que hereda implícitamente de la superclase del ***trait*** (en el ejemplo, será **AnyRef**) y
  - la mezcla con el ***trait***, heredando sus métodos.
- Si se quiere mezclar un ***trait*** en una clase que explícitamente extiende una superclase, se usa **extends** para indicar la superclase y **with** para mezclar en el ***trait***.

Traits (y2)

```
class Rectangulo extends Rectangular {
   …
}
```

Uso único de **extends**

```
class Rectangulo extends Figura with Rectangular {
   …
}
```

Uso de **extends** y **with**

Ejemplo completo figuras geométricas

```
abstract class Figura {
   val nombre: String
   def print(): Unit
   def DrawBoundingBox():Unit
}
trait Rectangular {
   def coordSupIzq: (Int,Int)
   def coordInfDer: (Int,Int)
   def izq = coordSupIzq._1
   def der = coordInfDer._2
   def ancho = der - izq
}
class Rectangulo(val nombre:String, var
coordSupIzq:(Int,Int), var coordInfDer:
(Int,Int))
extends Figura with Rectangular {
   def print():Unit = {
      println("Rectangulo: "+ nombre)
      println("Coord 1: " + coordSupIzq)
      println("Coord 2: " + coordInfDer)
   }
   def DrawBoundingBox():Unit = {
      print
      println("BoundingBox: ")
      println("Punto Sup Izq: " + coordSupIzq)
      println("Punto Inf Der: " + coordInfDer)
   }
}
```

```
class Circulo(val nombre:String, var centro:
(Int,Int), var radio:Int) extends Figura {
   def print():Unit = {
      println("Circulo: " + nombre)
      println("Centro: " + centro)
      println("Radio: " + radio)
   }
   def DrawBoundingBox():Unit = {
      print
      println("BoundingBox: ")
      println("Punto Sup Izq:" + (centro._1-
radio, centro._2-radio))
      println("Punto Inf Der: "+
(centro._1+radio, centro._2+radio))
   }
}
object Geom extends Application {
   val rec = new Rectangulo("rec1",(5,5),
(45,67))
   val cir = new Circulo("cir1",(20,20),12)
   rec.izq
   rec.der
   rec.ancho
   rec.DrawBoundingBox
   cir.DrawBoundingBox
}
```

Ejemplo completo figuras geométricas (y1)

- Lo grabamos en el fichero Geom.scala y lo probamos:

```
> scalac Geom.scala
> scala Geom
Rectangulo: rec1
Coord 1: (5,5)
Coord 2: (45,67)
BoundingBox:
Punto Sup Izq: (5,5)
Punto Inf Der: (45,67)
Circulo: cir1
Centro: (20,20)
Radio: 12
BoundingBox:
Punto Sup Izq: (8,8)
Punto Inf Der: (32,32)
```

Ejemplo queue

```
import scala.collection.mutable.ListBuffer

abstract class IntQueue{
   def get(): Int
   def put(x: Int)
}

class BasicIntQueue extends IntQueue {
   private val buf = new ListBuffer[Int]
   def get() = buf.remove(0)
   def put(x:Int) = buf += x
}
```

```
scala> val cola = new BasicIntQueue
scala> cola.put(10)
scala> cola.put(20)
scala> cola.get()
res2: Int = 10
scala> cola.get()
res3: Int = 20
```

A queue is a first-in, first-out (FIFO)

Ejemplo queue con traits

- Ahora vamos a usar ***traits*** a modificar su comportamiento (vamos a hacer modificaciones apilables).

```
trait Doubling extends IntQueue {
   abstract override def put(x:Int) = super.put(2 * x)
```

El ***trait Doubling*** hace dos cosas muy interesantes:

- Declara como superclase IntQueue. Esta declaración hace que el trait sólo pueda ser mezclado en una clase que también extienda IntQueue.

- El trait tiene una llamada super a un **método declarado abstracto**. En una clase normal sería ilegal, pero es posible en un ***trait***; los modificadores **abstract override** (sólo admitidos en los métodos de los ***traits***) indican que el trait se debe mezclar en alguna clase que tenga la definición concreta del método.

Ejemplo queue con traits (y1)

```
class MyQueue extends BasicIntQueue with Doubling
```

```
scala> val cola = new MyQueue
scala> cola.put(10)
scala> cola.get()
res2: Int = 20
```

La clase MyQueue no define código nuevo, sólo mezcla una clase con un ***trait***. En esta situación, se podría hacer el **new** directamente.

```
scala> val cola = new BasicIntQueue with Doubling
scala> cola.put(10)
scala> cola.get()
res2: Int = 20
```

https://www.cs.helsinki.fi/u/wikla/OTS/Sisalto/examples/html/ch12.html

Ejemplo queue con traits (y2)

- Añadimos dos nuevos ***traits***:

```
trait Incrementing extends IntQueue {
   abstract override def put(x:Int) = super.put(x + 1)
}
```

Definimos una cola que filtra números negativos

```
trait Filtering extends IntQueue {
   abstract override def put(x:Int) = if (x >= 0) super.put(x)
}
```

Añade uno a todos los números que almacena.

```
scala> val cola = new BasicIntQueue with Incrementing with Filtering
scala> cola.put(-1)
scala> cola.put(0)
scala> cola.put(1)
scala> cola.get()
res2: Int = 1
scala> cola.get()
res3: Int = 2
```

Mixins

- El orden de los ***mixins*** (cuando un trait se mezcla con una clase, se le puede llamar ***mixin***) es significante.
- **Van de derecha a izquierda**.
- Cuando se llama a un método de una clase con ***mixins***, se llama primero al método del ***trait*** más a la derecha. Si este método a su vez se llama **super**, invocará al método del siguiente ***trait*** a su izquierda.

Ejemplo queue con traits (y2)

- Añadimos dos nuevos ***traits***:

```
trait Incrementing extends IntQueue {
   abstract override def put(x:Int) = super.put(x + 1)
}
trait Filtering extends IntQueue {
   abstract override def put(x:Int) = if (x >= 0) super.put(x)
}
```

```
scala> val cola = new BasicIntQueue with Incrementing with Filtering
scala> cola.put(-1)
scala> cola.put(0)
scala> cola.put(1)
scala> cola.get()
res2: Int = 1
scala> cola.get()
res3: Int = 2
```

1) Definimos una cola que filtra números negativos

2) Añade uno a todos los números que almacena.

Mixins (y1)

- En el ejemplo anterior, primero se invoca al put de Filtering, de forma que se eliminan los enteros negativos y se añade uno a aquellos que no se han eliminado.
- Si invertimos el orden:

```
scala> val cola = new BasicIntQueue with Filtering with Incrementing
scala> cola.put(-1)
scala> cola.put(0)
scala> cola.put(1)
scala> cola.get()
res2: Int = 0
scala> cola.get()
res3: Int = 1
scala> cola.get()
res3: Int = 2
```

1) Se incrementarán los enteros
2) Se eliminarán los enteros que todavía siguen siendo negativos.

Jerarquía de clases

Jerarquía de clases en Scala

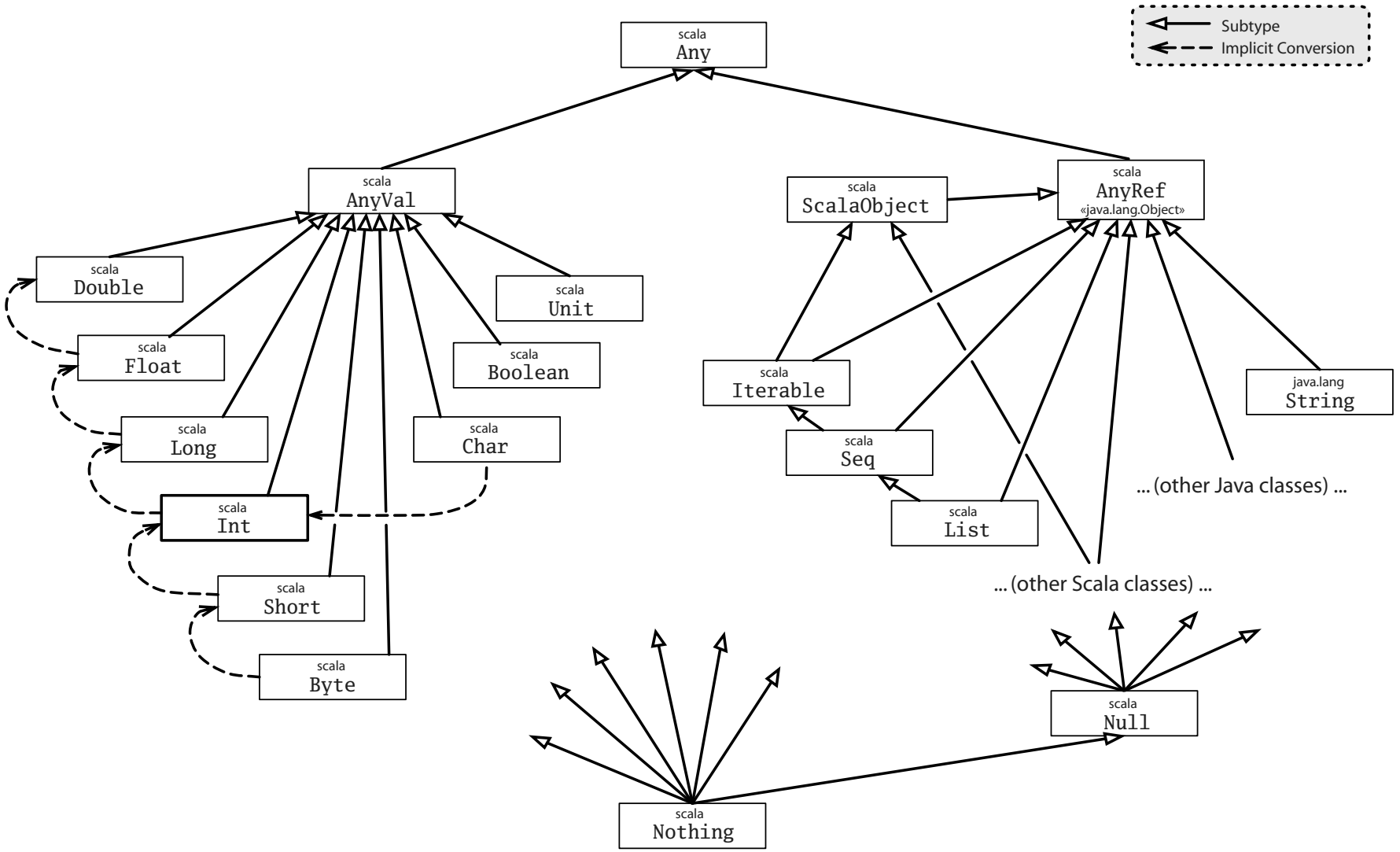
La superclase de todas las clases, **scala.Any**

Tiene dos subclases directas:

- **Scala.AnyVal** clases para valores.
  - Todas las clases para valores están predefinidas;
  - se corresponden con los tipos primitivos de los lenguajes tipo Java.

- **scala.AnyRef** clases para referencias.
  - Todas las otras clases definen tipos referenciables.

Jerarquía de clases en Scala



Jerarquía en Scala: Clase Any

- La clase Any define los siguiente métodos:
  - final def ==(that: Any): Boolean
  - final def !=(that: Any): Boolean
  - def equals(that: Any): Boolean
  - def hascode: Int
  - def toString: String

¿Se puede hacer override en las subclases de “==“ y “!=“?

¡¡Todo objeto en Scala puede usar estos métodos!!

Jerarquía en Scala: Clase AnyVal

- La clase AnyVal define los tipos de variable de Scala:
  - Byte, Short, Char, Int, Long, Float, Double, Boolean y Unit.
  - se corresponden con los tipos primitivos de los lenguajes tipo Java

42 es una instancia de Int
x es una instancia de Char
¡¡No se pueden crear instancias de estas clases usando new!!

Int tiene los métodos +, *.
Boolean tiene los métodos || y &&

Jerarquía en Scala: Clase AnyRef

- Esta es la clase base de todas las clases de referencia de Scala.
- Las clases definidas por el usuario son definidas como tipos referenciables por defecto, es decir, siempre (indirectamente) extienden de scala.AnyRef.
- Toda clase definida por usuario en Scala extiende implícitamente el trait scala.ScalaObject
  - Por ejemplo, en Java, esta clase es un alias para la clase java.lang.Object.

Jerarquía en Scala: Clases Null y Nothing

- ***Null***: Representa el tipo referencia nula. Se utiliza si es necesario utilizar código Java.

```
scala> val i: Int = null
<console>:4: error: type mismatch;
   found   : Null(null)
   required: Int
```

No podemos asignar el valor null a una variable.

- ***Nothing***: No tiene valores. Se suele utilizar para métodos que nunca terminan de forma correcta (se lanza una excepción). Por ejemplo:

```
def error(message: String): Nothing =
   throw new RunTimeException(message)
```

Funciones y Closures

Funciones como objetos de primera clase

- En Scala las funciones son también objetos de primera clase.
- **Las funciones son de primer orden, cuando pueden pasarse como argumento o ser devueltas como resultado por otra función**
- Podemos:
  - Definir variables y parámetros de tipo función.
  - Almacenar funciones en estructuras de datos como listas o arrays.
  - Construir funciones en tiempo de ejecución (closures) y devolverlas como valor de retorno de otra función.

Ejemplo: sumatorio

- Scala es un lenguaje tipado y hay que definir el tipo de la función que se pasa como parámetro.

La función **f**, que se pasa como primer parámetro de sum, debe recibir un entero y devolver un entero.

```
def cuadrado(x: Int): Int = x * x

def sum(f: Int => Int, a:Int, b:Int): Int =
   if (a>b) 0 else f(a) + sum(f, a+1, b)

sum(cuadrado _, 1,9)
```

Funciones anónimas

- En Scala se pueden definir funciones anónimas creadas en tiempo de ejecución.

```
(x: Int) => x * x
```

Definición de la función **cuadrado** anterior de ***forma anónima***.

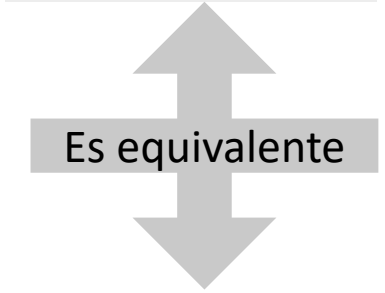
```
sum(x => x * x, a, b)
```

Podemos definirla como parámetro de **sum** y no hace falta definir el tipo de las variables, ya que Scala infiere el tipo.

Funciones anónimas (y1)

- Una notación más concisa utiliza subrayados como huecos (***placeholders***) de los parámetros de la función.

```
sum(_+2, a, b)
```




```
sum(x => x+2, a, b)
```

Variables de tipo función

- Podemos asignar funciones a variables:

```
val f = (s:String) => s + “adios”

f(“hola”)
```

- Para asignar una función ya definida a una variable hay que utilizar la sintaxis de huecos (placeholders) para indicar al compilador que no tiene que aplicar la función:

```
def suma3(a: Int, b: Int, c: Int) = a + b + c

val f = suma3 _

f(1,2,3) --> 6
```

Variables de tipo función (y1)

- Podemos almacenar funciones en listas, pero todas las funciones deben tener el mismo perfil:

```
def suma3(a: Int, b: Int, c: Int) = a + b + c
def mult3(a: Int, b: Int, c: Int) = a * b * c

val listaFuncs = List(suma3 _, mult3 _)

val f = listaFuncs.head

f(1,2,3)
```

Variables de tipo función (y2)

- Ejemplo de función que toma una lista de funciones de un argumento y las aplica:

```
def aplicaLista (lista: List[(Int) => Int], x: Int): Int =
   if (lista.length == 1) lista.head(x) else
      lista.head(aplicaLista(lista.tail,x))

def mas5(x: Int) = x+5
def por8(x: Int) = x*8

val l = List(mas5 _, por8 _)

aplicaLista(l, 10)
```

Variables de tipo función (y3)

- Es posible utilizar la sintaxis de los huecos para proporcionar algunos de los argumentos y dejar libres otros.

```
val g = suma3(1, _: Int, 10)
```

Definición de una variable de tipo función de un argumento a partir de una función anterior

Clausuras

- Las funciones definidas en un ámbito mantienen el acceso a ese ámbito y pueden usar los valores allí definidos.
- Llamamos ***clausura*** a estas funciones

```
def makeClosure(i : Int) : Int => Int = {
   val x = 5
   (j : Int) => i + j + x
def prueba(y: Int): (Int)=>Int = {
val x = 0
(z:Int) => {x+y+z} }
val f = prueba(2) f(3)
}

val returnedFuncion : Int => Int = makeClosure(10)
println(returnedFuncion) //=> <function>
println(returnedFunction(20)) //=> 35, porque 10 + 20 + 5 = 35
```

El ámbito en el que se crea la función returnedFunction contiene la variable i (el argumento de makeClosure) y la variable valor x declarada con el valor 5.

Ejemplos con Clausuras

```
def f(x: Int, y: Int): Int = {

val z = 5
x+y+z
}

 def g(z: Int): Int = {
 val x = 10
z+x
}


f(g(3),g(5))
```

1) g(3)
X=10
Z=3 =>Z=13

2) g(5)
X=10
Z=5 =>Z=15

3) f
X=13
Y=15
Z=5 =>X+Y+Z=33

El valor de f()=33

Ejemplos con Clausuras

```
val x = 10
val y = 20

def g(y: Int): Int = {
      x+y
}

def prueba(z: Int): Int = {
val x = 0
g(x+y+z)
}
val f = prueba(3)
```

1) X=10
   Y=20

3) Y=23
   X=10 => x+y= 23+10

2) Z=3
   X=0
   Y=20 => x+y+z= 0+20+3

El valor de g()=33

Ejemplos con Clausuras

```
val x = 10
val y = 20

def prueba(y: Int): (Int)=>Int = {
val x = 0
(z:Int) => {x+y+z}
}

val f: (Int) => Int = prueba(10)
f(30)
```

X=10

Y=20

Y=10

X=0 => f(z)

z=30

X=0 y=10 => f(z) x+y+z=
0+10+30 = 40

El valor de f(30)=40

Ejemplos con Clausuras

```
val x = 100
def func1(): () => Int = {
val x=1
() => {x+1}
}
def func2(): Int= {
val x=10
val f=func1()
f()
}
func2 ()
```

```
x = 100

func2() {
        x=10
        f()
}
       func1(){
         x=1
          f= {x+1}
f()
}
f (x+1, donde x=1  f()=2)
```

El valor de f()=2

Tail recursion (Recursión de cola)

- Una función es ***recursiva de cola*** si la llamada recursiva es la última instrucción que se ejecuta, en cualquier circunstancia.

```
def collatz(n : Int) : Int = {
   if (n == 1) 1
   else if (n % 2 == 0) collatz(n / 2)
   else collatz(3 * n + 1)
}
```

- Si no hay recursión: n = 1
- En cualquier otro caso, la última instrucción que se ejecuta es la llamada recursiva.

```
def collatz(n : Int) : Int = {
   var nn = n
   while (nn != 1) {
      if (nn % 2 == 0) nn = nn / 2)
      else nn = 3 * nn + 1
   }
   nn
}
```

Un buen compilador reemplaza la recursión de cola con un bucle.

Tail recursion (Recursión de cola)

- Una función es ***recursiva de cola*** si la llamada recursiva es la última instrucción que se ejecuta, en cualquier circunstancia.

```
def factorial(n: BigInt): BigInt = {
 def go(acc: BigInt, n: BigInt): BigInt = {
         if (n <= 1)
           acc
         else
           go(acc * n, n - 1) }
go(1, n) }
```

```
// tail-recursive solution
def sum(list: List[Int]): Int = {
  def sumWithAccumulator(list: List[Int], currentSum: Int): Int = { list match {
      case Nil => currentSum
      case x::xs => sumWithAccumulator(xs,currentSum + x)
     }
}
sumWithAccumulator(list, 0) }
```

Currying

Currying

Es la técnica de transformación una función que tiene múltiples argumentos (tupla of argumentos) de tal manera que se le puede llamar como una cadena de funciones cada uno como un único argumento (aplicación parcial).

```
def suma (x: Int, y: Int) = x + y

val s = suma(5, 12)
```

Llamamos a la función

- Suponemos cómo el compilador sustituye cada variable con un valor:

```
suma(x + y) → suma(5 + y)
suma(5 + y) → suma(5 + 12)
```

Paso1: Sustituimos ‘x’ por 5.

Paso2: Sustituimos ‘y’ por 12.

- Tras el Paso1, tenemos una **función aplicada parcialmente**.
- ***Currying*** es una forma de realizar funciones aplicadas parcialmente.

Currying en Scala

- En lugar de tener la función **suma** con una lista de parámetros, podemos tenerla con dos listas de parámetros:

```
scala> def suma(x: Int) (y: Int) = suma (x + y)
suma: (x: Int)(y: Int)Int

scala> suma(5)(12)
res1: Int = 17

scala> val s2 = suma(5)_
h2: (Int) => Int = <function1>
scala> s2(12)
res2: Int = 17
```

Llamamos a la función, en lugar de una lista de dos argumentos, usamos dos listas de argumentos.

Es necesario utilizar el símbolo '_' para decir a Scala que hay un parámetro no definido.

```
def foo(a: Int)(b: Int, c: String)(d: Double) = { … }
```

Currying en Scala (y1)

```
scala> def media(a: Double, b: Double) = (a + b) / 2.0
media: (a: Double, b: Double)Double


scala> media(5)(12)
res3: Double = 8.5


scala> val m5 = media(5, _: Double)
a5: (Double) => Double = <function1>


scala> m5(12)
res4: Double = 8.5
```

Llamamos a la función

Llamamos a la función “currificada”

Llamamos a la función especificando el primer parámetro y el segundo sin especificar (‘_’)

- Hay que especificar el tipo del parámetro omitido.
- Atención al tipo del resultado de la función.

Colecciones

Colecciones en Scala

Las colecciones o contenedores de objetos están presentes en todos los programas y todos los lenguajes

Se distinguen sistemáticamente entre colecciones **mutables o inmutables**

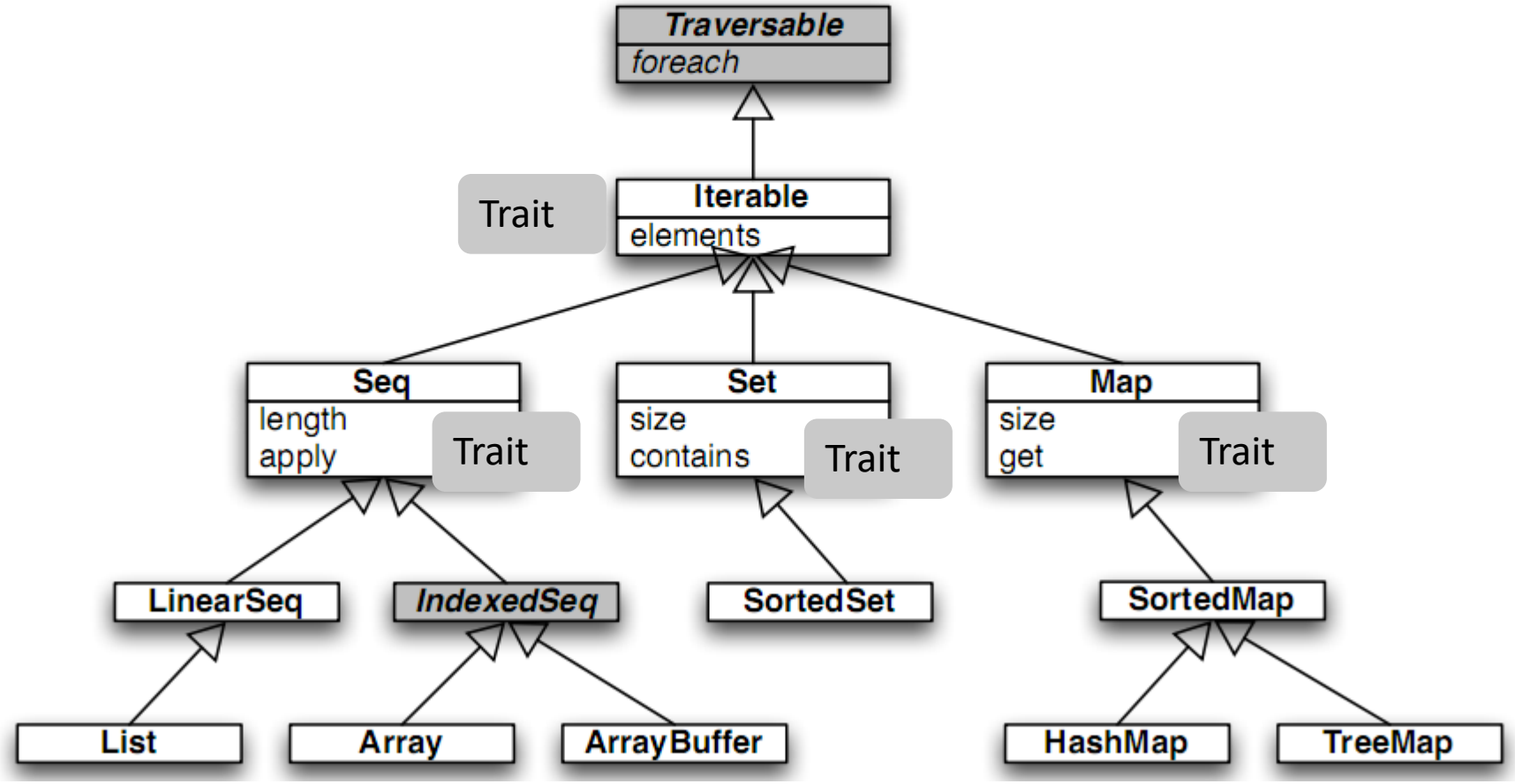
- **colección mutable** se puede agregar, cambiar o remover elementos de una colección .
- **colección inmutables**, nunca cambian, puede haber operaciones que simulen ediciones, agregación , o eliminación, pero estas operaciones en cada caso retornaran una nueva colección y dejaran sin cambiar la antigua.

Jerarquía de la Arquitectura colecciones

Son todos los rasgos **(traits),** tanto los paquetes mutables e immutables tienen implementaciones de éstos así como también implementaciones especializadas.

- **Recorrible (Traversable)**
  - Todas las colecciones pueden ser recorribles.
  - Estos recorridos están escritos en términos de foreach, el cúal las colecciones deben implementar.
- **Iterable (Iterable)**
  - Tiene un método iterator() que devuelve un Iterador para recorrer los elementos.
- **Secuencia (Seq)**
  - Secuencia de objetos con ordenamiento.
- **Conjunto (Set)**
  - Una colección de objetos no duplicados.
- **Mapa (Map)**
  - Pares de claves con valores.

Colecciones en Scala



Secuencias

- Permiten trabajar con grupos de elementos ordenados.
- Las más importantes son:
  - List
  - Array (empiezan en el índice 0).
  - ListBuffer. Permite construir listas más eficientes a la hora de añadir elementos.

```
scala> val buf = new ListBuffer[Int]
scala> buf += 1          // ListBuffer(1)
scala> buf += 2          // ListBuffer(1,2)
scala> 3 +: buf          // ListBuffer(3,1,2)
scala> buf.toList        //List(3,1,2)
```

Secuencias (y1)

- Las más importantes son (cont.)
  - ArrayBuffer. Permite añadir/eliminar elementos.
  - Queue. Permite estructura FIFO.

```
scala> val queueI = new Queue[Int]
scala> val has1 = queueI.enqueue(1) //Añadir un elemento a la cola
scala> val has123 = queueI.enqueue(List(2,3))  //queueI = (1,2,3)
scala> val (element, has23) = has123.dequeue   //queueI = (2,3)

scala> val queueM = new Queue[String]
scala> queueM += “a”  //queueM = (a)
```

Queue Inmutable

Queue Mutable

- Añadir elementos: +=, ++=
- Sacar elementos: dequeue

Secuencias (y2)

- Las más importantes son (cont.)
  - Stack. Permite estructura LIFO.

```
scala> val stack = new Stack[Int]
scala> stack.push(1)  //Añadir un elemento
scala> stack.push(2)  // stack = (1,2)
scala> stack.top      // 2
scala> stack.pop      // Sacar el elemento 2
scala> stack          // stack = (1)
```

  - String.

Conjuntos

- Scala provee operaciones de conjuntos a través de la clase **Set**.
- Scala permite trabajar con conjuntos mutables o inmutables. Por defecto, Scala trabaja con conjuntos inmutables.

```
var conjunto = Set[Int]()
```

Se debe especificar el tipo de un conjunto vacío

```
var conjunto = Set(1, 3, 5, 7, 11)
```

Conjunto con elementos: el compilador infiere el tipo de datos

Si se quiere trabajar con conjuntos mutables …

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

- Las operaciones más comunes con conjuntos son + y +=.

Conjuntos (y1)

```
var trilogias = Set(“El Padrino, “El Señor de los Anillos”)
trilogias += “Millenium”
println(trilogias)
```

- ¿Conjunto Mutable?
- Se crea un nuevo conjunto “trilogias” con el valor “Millenium”

```
import scala.collection.mutable.Set
val trilogias = Set(“El Padrino, “El Señor de los Anillos”)
trilogias += “Millenium”
println(trilogias)
```

- Se añade el elemento “Millenium” al conjunto “trilogias”

Conjuntos(y2)

a **diff** d //1,2,5,Nil
a **--** b //1,2, 5,Nil
a **--** a //¿Salida?

d **contains** 3//True

a **intersect** b //3, 4
a **&** b //3, 4
a ** b //Versiones anteriores
a & c //¿Salida?

```
var a = Set(1,2,3,4)
var b = Set(3,4,5,6)
var c = Set(1,2,"hola")
var d = Set(3,4,5,Nil)
var e = Set(1,2)
var f = Set[Int]()
```

e **subsetOf** a //True
f **subsetOf** a //¿Salida?

a **isEmpty** //False
f **isEmpty** //¿Salida?

c **union** d //1,2,"hola",3,4,5,Nil
a **|** b //1,2,"hola",3,4,5,Nil
a **++** b //¿Salida?
Array(1,3)++Array(3,4) //¿Salida?

Conjuntos (y3)

- Conjuntos por defecto:

Número de elementos	Implementación
0	scala.collection.inmutable.EmptySet
1	scala.collection.inmutable.Set1
2	scala.collection.inmutable.Set2
3	scala.collection.inmutable.Set3
4	scala.collection.inmutable.Set4
5 o más	scala.collection.inmutable.HashSet

Mapas

- Scala tiene soporte para mapas a través de dos implementaciones para la clase Map:
  - Mutable
  - Inmutable
- Los mapas son contenedores asociativos para los que se definen parejas de llave-valor.
- Las llaves en Scala pueden ser cualquier objeto, usualmente son cadenas.

```
val m = Map(
   “hola” -> List(‘h’, ‘o’, ‘l’, ‘a’),
   “hi” -> List(‘h’, ‘i’))

println(m(“hola”))
```

- “hi” → Llave
- -> → Flecha derecha
- List(‘h’, ‘i’) → Valor
- **Mapa inmutable**

Mapas (y1)

```
/* mapas.scala */
import scala.collection.mutable.Map
val m = Map[Int, String]()
m += 1 -> “uno”
m += ((4, “cuatro”))

println(m(1))
```

- Creación de un mapa vacío →Definir tipo

```
$ scala mapas.scala

"uno"
```

Mapas (y2)

Método	Resultado
+=	Asigna/Agrega una nueva pareja al mapa
contains	Verifica si una llave está presente en un mapa
-=	Elimina una pareja del mapa especificando la llave de la misma
keys	Obtiene una colección iterable con las llaves del mapa
values	Obtiene una colección iterable con los valores del mapa

Mapas (y3)

- Mapas por defecto:

Número de elementos	Implementación
0	scala.collection.inmutable.EmptyMap
1	scala.collection.inmutable.Map1
2	scala.collection.inmutable.Map2
3	scala.collection.inmutable.Map3
4	scala.collection.inmutable.Map4
5 o más	scala.collection.inmutable.HashMap

Tuplas

- Las tuplas son secuencias ordenadas de objetos, similares a las listas en el sentido de que son inmutables.
- Las tuplas en Scala se utilizan para agrupar de manera ordenada objetos, que en general son de tipos diferentes.
- El uso ideal de una tupla es cuando se requiere retornar múltiples objetos en la invocación de un método.
- Para instanciar una tupla, simplemente se colocan los objetos que la conforman entre paréntesis, separados por comas:

```
var tupla1 = (1, “uno”, List(1))
```

- Para acceder a un elemento de la tupla se hace de la siguiente forma:

```
tupla1._1 //Primer elemento
tupla1._3
```

Tuplas (y1)

- El tipo de una tupla depende del número de argumentos y el tipo de cada uno de ellos.

```
var tupla1 = (1, “uno”)
var tupla2 = (‘s’, ‘a’, 30, List(1))
```

Tipo Tuple2[Int, String]

Tipo Tuple4[Char, Char, Int, List[Int]]

- Se pueden definir actualmente tuplas de hasta 22 elementos en Scala.

Funciones de orden superior

La definición de funciones como un tipo de dato primitivo permite definir funciones de orden superior que toman como parámetro otras funciones y permiten hacer mucho más conciso y general el código escrito en Scala.

Ejemplos de métodos de orden superior de la **clase List**:

- **count** --> cuenta el numero de elementos de la lista que satisfacen un predicado
- **exists** --> comprueba si algún elemento de la lista satisface un predicado
- **filter** --> devuelve una nueva lista con los elementos de la lista original que cumplen el predicado
- **map** --> aplica una función a todos los elementos de la lista, devolviendo una nueva lista con el resultado

Son métodos y no funciones

Ejemplos de orden superior ()

```
val lista = List(1,2,3,4,5,6,7,8,9)
lista.count((x:Int) => {x % 2 == 0}) // Int: 4
```

```
lista.count((x) => {x % 2 == 0})
lista.count(x => x % 2 == 0)
lista.count(_ % 2 == 0)
def par(x:Int): Boolean = {x % 2 == 0}
 lista.count(par _)
```

```
lista.filter(par _)        // List (2,4,6,8)
def cuadrado(x:Int) = x*x
lista.map(cuadrado _)     // List (1,4,9,16,25,36,49, 64,81)
List(“esto","es",”una",”prueba).map(s => s.toUpperCase)
// List(ESTO, ES, UNA, PRUEBA)
```

Parametros por Valor y por nombre

Normalmente los parámetros de las funciones son parámetros **por-valor**.
Significa que el valor del parámetro se determina antes de que se pase a la función

Pero a veces puede que necesitemos escribir una función que acepte como parámetro una expresión que no queremos que se evalúe hasta que sea llamada nuestra función. **Por nombre.**

Por valor

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
9+4*4
9 + 16
25
```

Por Nombre

```
sumOfSquares(3, 2+2)
square(3) + square(2+2)
3 * 3 + square(2+2)
9 + square(2+2)
9 + (2+2) * (2+2)
9+4*(2+2)
9+4*4
9 + 16
25
```

Packages e Imports

Packages e Imports

- Revisar la siguiente dirección

http://www.artima.com/pins1ed/packages-and-imports.html

Case classes and pattern matching

Case Classes y Pattern Matching

- Revisar la siguiente dirección

http://www.artima.com/pins1ed/case-classes-and-pattern-matching.html

Abstract members

Abstract Members

- Revisar la siguiente dirección

http://www.artima.com/pins1ed/abstract-members.html