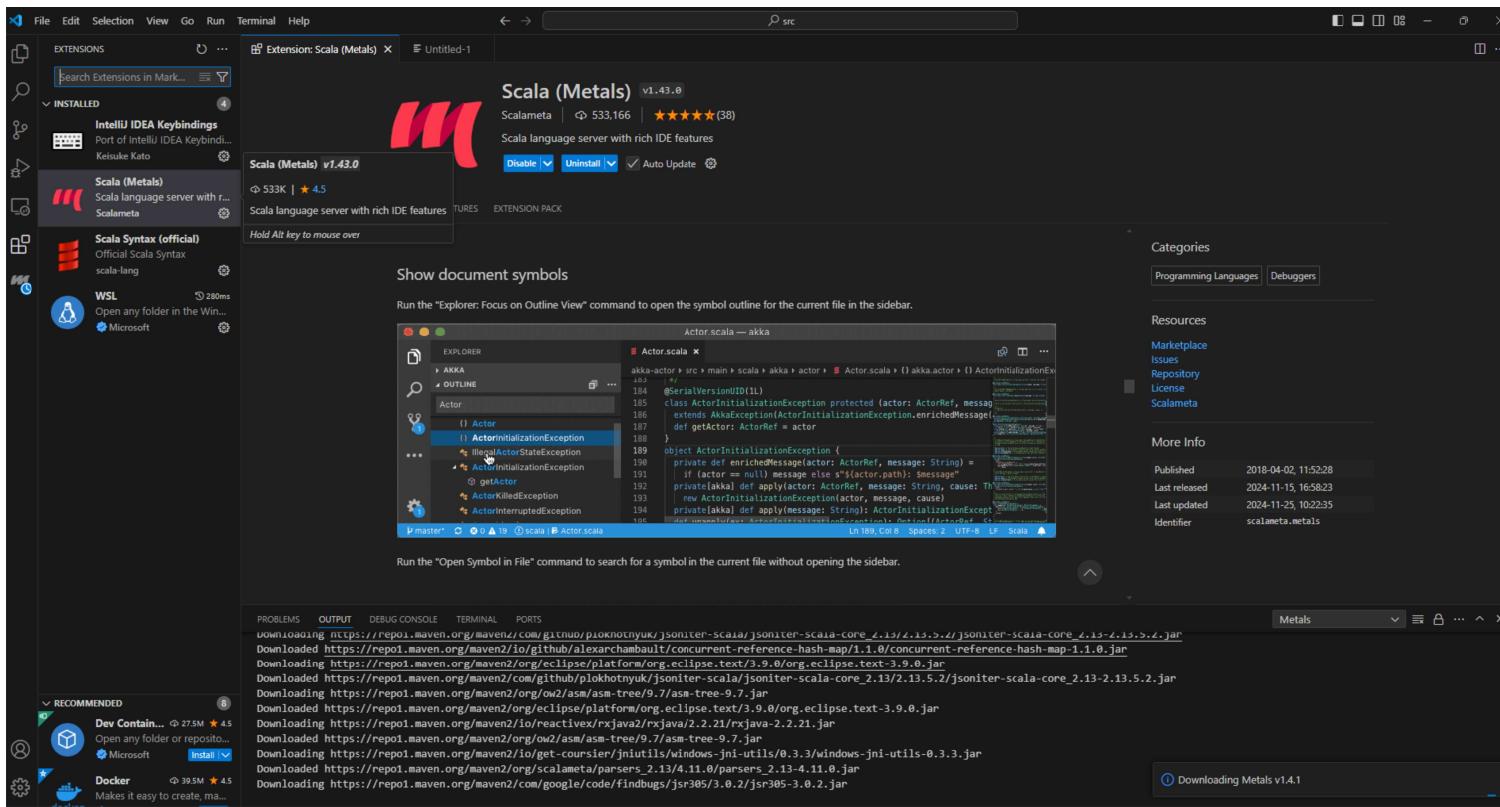




## Módulo Profesional: Big Data Aplicado

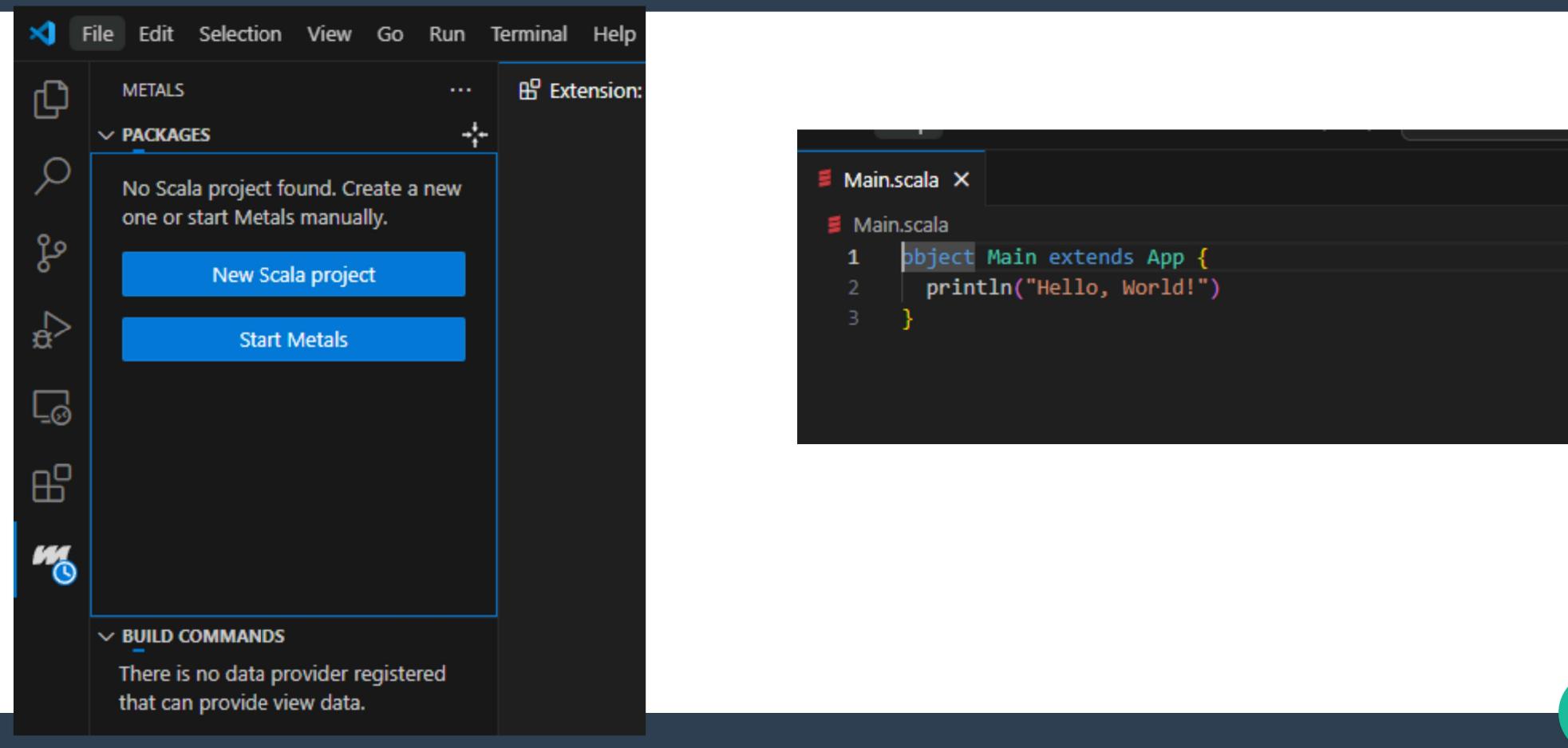
Scala

# **COMENZAR SCALA: Visual Studio Code**



2

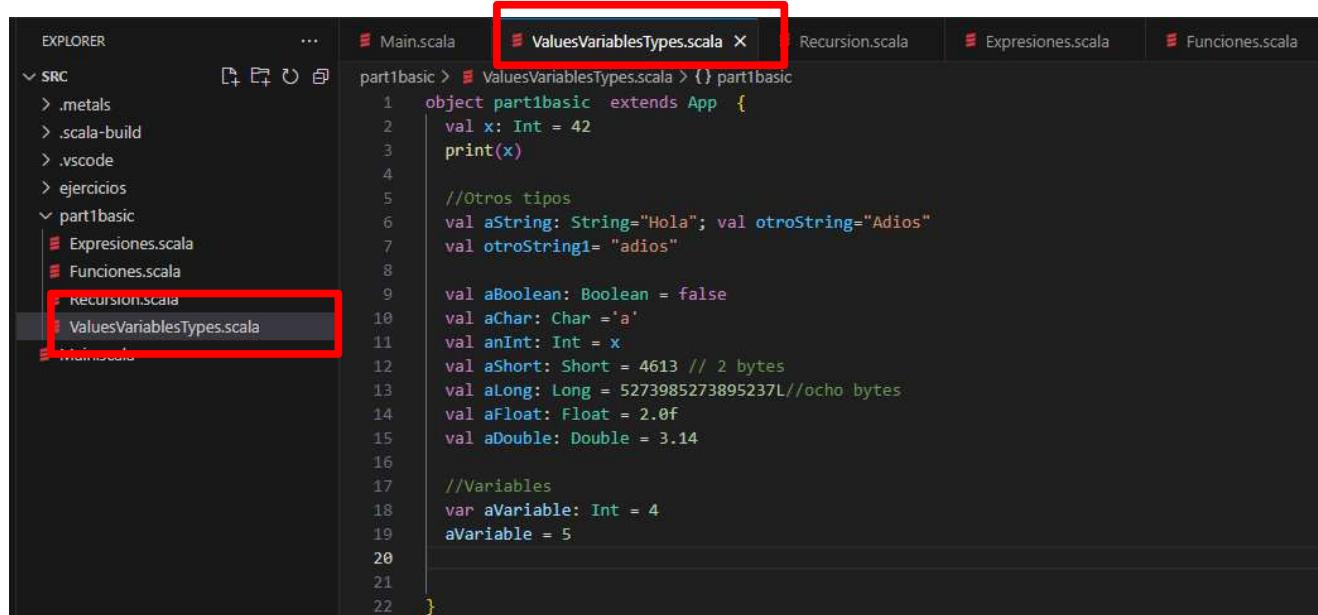
# COMENZAR SCALA: Visual Studio Code



# VALUES, VARIABLES Y TYPES

Nuevo File Scala de tipo Object llamada ValuesVariablesTypes.

Añadimos **extends App**



```
object part1basic extends App {  
    val x: Int = 42  
    print(x)  
  
    //Otros tipos  
    val aString: String = "Hola"; val otroString = "Adios"  
    val otroString1 = "adios"  
  
    val aBoolean: Boolean = false  
    val aChar: Char = 'a'  
    val anInt: Int = x  
    val aShort: Short = 4613 // 2 bytes  
    val aLong: Long = 5273985273895237L // ocho bytes  
    val aFloat: Float = 2.0f  
    val aDouble: Double = 3.14  
  
    //Variables  
    var aVariable: Int = 4  
    aVariable = 5  
}
```

# VALUES (VAL) , VARIABLES (VAR) Y TIPOS (TYPES)

```
//Ejemplo 1
val x: Int = 42
println(x)

x = 2
```

## Regla:

- “val” no puede ser reasignado.
- ES INMUTABLE
- Se comporta de forma similar a las constantes de Java o C, aunque tienen propósito diferente.

Scala nuevo paradigma de programación funcional que implica en general trabajar con **val**.  
¡A medida que avancemos lo verás!

# VALUES (VAL) , VARIABLES (VAR) Y TIPOS (TYPES)

```
//Ejemplo 2  
val y = 42  
println(y)
```

**“val” inferir el tipo, es opcional indicarlo.**

El compilador comprueba el valor derecho 42 que es un entero por lo que deduce que “y” es entero.

# VALUES (VAL) , VARIABLES (VAR) Y TIPOS (TYPES)

```
//Ejemplo 3  
val t: Int = "Hola"  
println(t)
```

¿Funciona?

```
//Ejemplo 4  
val aTexto: String = "Hola"; val aTexto1 = "adios"
```

; sólo es necesario si escribes múltiples expresiones en la misma línea

```
//Ejemplo 4  
val aTexto: String = "Hola"  
| val aTexto1 = "adios"
```

# VALUES (VAL) , VARIABLES (VAR) Y TIPOS (TYPES)

```
//Ejemplo 5: Otros tipos
val aBoolean: Boolean = false
val aCaracter: Char = 'a'
println(aCaracter)
val aEntero: Int = x
println(aEntero)
```

```
val aShort: Short = 46135713 |
```

¿Funciona?

# VALUES (VAL) , VARIABLES (VAR) Y TIPOS (TYPES)

```
val aShort: Short = 4613
val aLong: Long = 5273985273895237L
val aFloat: Float = 2.0f
val aDouble: Double = 3.14
```

# VALUES (VAL , VARIABLES (VAR) Y TIPOS (TYPES)

```
//VARIABLES  
var aVariable: Int = 4  
aVariable = 5
```

¿Funciona?

```
val x: Int = 42
```

Inmutable = no se puede reasignar

```
var x: Int = 1  
x = 1  
x += 1
```

Mutable

# EXPRESIONES

## Ejemplos de Expresiones

```
//1. Expresión  
val x = 1 + 2  
println(x)  
  
//2. Expresión  
println(2 + 3 * 4) // + - * /  
  
//3. Expresión, se evalua como boolean  
println (1 == x) //== != > >= < <=  
  
//4. Expresión  
println(!(1 == x)) //! && || operadores lógicos  
  
//5. Expresión  
var aVariable = 2  
aVariable += 3 // también podemos trabajar con -= *= /=  
println(aVariable) // el valor devuelto será 5
```

# EXPRESIONES

## Instrucciones (DO) vs Expresiones (VALUE)

```
//Instrucciones (algo que haga la computadora) vs Expresiones (algo que tiene un valor)

val aCondicion = true
val aCondicionValor = if (aCondicion) 5 else 3 // IF como expresión no como instrucción
println(aCondicionValor)
println(if(aCondicion) 5 else 3)
println(1 + 3)
```

# EXPRESIONES

## Instrucciones (DO) vs Expresiones (VALUE)

```
var i = 0
while ( i < 10) {
    println (i)
    i += 1
}
```

# EXPRESIONES

## Instrucciones (DO) vs Expresiones (VALUE)

```
var i = 0
while ( i < 10) {
    println(i)
    i += 1
}
```

Nunca escribas esto otra vez  
Todo en Scala es una Expresión

# EXPRESIONES

## Bloques de Código

```
//Bloques de código
val aCodBloque = {
    val t = 2
    val z = t + 2

    if (z > 2) "hola" else "adiós"
}
```

OJO: Val declarados dentro del bloque de código  
No son accesibles desde fuera del bloque.

¿aCodBloque type?

# EJERCICIOS

1.- Cuál es la diferencia entre “Hola” y `println (“Hola”)`

2. ¿Cuál es el tipo?

```
val tipoValor = {  
    2 < 3  
}
```

3. ¿Tiene valor?

```
val otroValor = {  
    If (tipoValor) 239 else 986  
    42  
}
```

# FUNCIONES

```
def aFuncion1(a: String, b: Int): String = {  
    a + " " + b  
}  
  
println(aFuncion1("Hola", 3))
```

```
def aParametrosFuncion(): Int = 42  
println(aParametrosFuncion())  
//println(aParametrosFuncion)
```

# FUNCIONES

```
//cuando necesites bucles, usa recursividad
def aRecursivaFuncion1(aString: String, n: Int): String = {
  if (n == 1) aString
  else aString + aRecursivaFuncion1(aString, n - 1)
}

println(aRecursivaFuncion1("Hola", 3))
```

# FUNCIONES

```
def principalFuncion (n: Int): Int = {  
    def secundariaFuncion (a: Int, b: Int): Int = a + b  
    secundariaFuncion(n, n - 1)  
}  
}
```

¿Puedes explicar qué hace esta función?

# EJERCICIOS

## Ejercicio 1.

Función de saludar function (nombre, edad) => "Hola, mi nombre es \$nombre y tengo \$edad años."

## Ejercicio 2

Función factorial  $1 * 2 * 3 * 4 * 5 * \dots * n$

## Ejercicio 3

Función de test si el número es primo

# Type Inference

¿Cómo sabe el compilador el tipo qué debe de Inferir?

val mensaje = **“Hola”**

Debe ser cadena

El compilador mira la parte derecha  
de la asignación y comprueba el tipo  
qué es:  
Esto es una cadena

El compilador infiere el tipo por nosotros de “String”

val mensaje: **String** = “Hola”

# Type Inference

```
val x = 2  
val y = x + "Hola"
```

¿x de qué tipo es?  
¿y de qué tipo es?

```
def sucesora (x: Int) = x + 1
```

¿qué tipo devuelve la función?

```
val z: Int = "Hola"  
println(z)
```

¿qué tipo devuelve la función?

# Type Inference

```
val x = 2  
val y = x + "Hola"
```

¿x de qué tipo es?  
¿y de qué tipo es?

```
def sucesora (x: Int) = x + 1
```

¿qué tipo devuelve la función?

```
val z: Int = "Hola"  
println(z)
```

¿qué tipo devuelve la función?

# Recursividad

Escribe la siguiente función y observa qué pasa

```
def factorial (n: Int): Int =  
    if (n <= 1) 1  
    else {  
        println ("Factorial de " + n + "factorial de n - 1 " + (n - 1))  
        val resultado = n * factorial (n - 1)  
        println("Factorial de n" + n)  
  
        resultado  
    }  
    println (factorial (10))  
    println(factorial(5000))
```

# Recursividad

## Desbordamiento de pila

The screenshot shows a terminal window with the following content:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Filter (e.g. text, !exclude, \escape) Recursion
Factorial de 11/4factorial de n - 1 11/3
Factorial de 1173factorial de n - 1 1172
Factorial de 1172factorial de n - 1 1171
Factorial de 1171factorial de n - 1 1170
Exception in thread "main" java.lang.StackOverflowError
    at java.base/java.nio.ByteBuffer.position(ByteBuffer.java:1526)
    at java.base/java.nio.ByteBuffer.position(ByteBuffer.java:267)
    at java.base/sun.nio.cs.SingleByte.withResult(SingleByte.java:48)
    at java.base/sun.nio.cs.SingleByte$Encoder.encodeArrayLoop(SingleByte.java:243)
    at java.base/sun.nio.cs.SingleByte$Encoder.encodeLoop(SingleByte.java:276)
    at java.base/java.nio.charset.CharsetEncoder.encode(CharsetEncoder.java:585)
    at java.base/sun.nio.cs.StreamEncoder.implWrite(StreamEncoder.java:374)
    at java.base/sun.nio.cs.StreamEncoder.implWrite(StreamEncoder.java:361)
    at java.base/sun.nio.cs.StreamEncoder.lockedWrite(StreamEncoder.java:162)
    at java.base/sun.nio.cs.StreamEncoder.write(StreamEncoder.java:143)
```

# Llamada por Nombre o Llamada por Valor

```
def LlamadaporValor (x: Long): Unit = {
    println ("Por valor" + x)
    println ("Por valor" + x)
}
```

```
def LlamadaporNombre (x: => Long): Unit = {
    println ("Por nombre" + x)
    println ("Por nombre" + x)
}
```

```
LlamadaporValor(System.nanoTime())
LlamadaporNombre(System.nanoTime())
```

# Llamada por Valor

```
def LlamadaporValor (x: Long): Unit = {  
    println ("Por valor" + x)  
    println ("Por valor" + x)  
}
```

```
LlamadaporValor(System.nanoTime())
```

Devuelve el tiempo actual en nanosegundos. El resultado se calcula una vez antes de que la función comience a ejecutarse.

La función `LlamadaporValor` toma un argumento `x` que se evalúa inmediatamente cuando se llama a la función. Significa que el valor de `x` se calcula una vez, antes de que se ejecute el cuerpo de la función y el mismo valor se utiliza en todas las referencias a `x` dentro de la función.

# Llamada por Nombre

```
def LlamadaporNombre (x: => Long): Unit = {  
    println ("Por nombre" + x)  
    println ("Por nombre" + x)  
}
```

```
LlamadaporNombre(System.nanoTime())
```

**System.nanoTime()** no se evalúa inmediatamente cuando se realiza la llamada a la función. Si no que se evalúa cuando es referenciado en el cuerpo de la función.

La función como parámetro `x: => Long`, significa que el argumento se pasa como una expresión sin evaluar y cada vez que `x` es utilizado en la función la expresión asociada se evalúa nuevamente.

# Llamada por Valor / Llamada por Nombre

```
/*Ejercicio 1: Explica la diferencia de resultado de imprimePrimer*/  
  
def Infinito(): Int = 1 + Infinito()  
def imprimePrimer(x: Int, y : => Int) = println (x)  
  
imprimePrimer(Infinito(), 34)  
imprimePrimer(34, Infinito())
```

# Parámetros por defecto o por nombre

Pasas todos los parámetros de la función o nombras los parámetros

```
def guardaPicture(formato: String = "jpg", ancho: Int = 1920, altura: Int = 1080): Unit =  
    | println("guarda picture")  
  
guardaPicture(ancho = 800)  
guardaPicture("bmp")  
guardaPicture(800, 600)  
guardaPicture("bmp", 600, 800)  
guardaPicture(altura = 600, ancho = 800, formato = "bmp")
```

## Parámetros por defecto o por nombre

En este caso uno en la primera llamada pasamos sólo un parámetro y en la segunda llamada pasamos los dos parámetros.

```
def factorial (x: Int, acc: Int = 1): Int = {
    ...
}

val fact10 = factorial (10)
val fact11 = factorial (11,2)
```

# Recursividad

## Recursividad no final:

Después de calcular factorial ( $n - 1$ ) todavía tiene que multiplicar ese resultado por  $n$  antes de devolverlo.

Esto significa que cada llamada recursiva debe esperar el resultado de las llamadas siguientes para completar su cálculo.

- Consumo de memoria
- Menor optimización

# Recursividad

```
object Recursividad1 extends App{  
    def factorial (n: Int): Int =  
        if (n <= 1) 1  
        else {  
            println ("Ejecutando Factorial de " + n + "factorial de n - 1 " + (n - 1))  
            val resultado = n * factorial (n - 1)  
            println("El Factorial de n" + n)  
  
            resultado  
        }  
    println (factorial (10))  
    println(factorial(5000))
```

# Recursividad

## Recursividad final:

La llamada recursiva es la última operación que se realiza en cada invocación de factCalcula.

La función se llama así misma con los valores actualizados y no realiza ningún cálculo después de la llamada recursiva

- Eficiencia
- Optimización automática. El compilador puede optimizar recursiones finales para transformarlas en un simple bucle.

# Recursividad

```
import scala.annotation.tailrec
run | debug
object Recursividad2 extends App{

    //nuevo concepto --BigInt

    def otroFactorial(n: Int): Int = {
        def factCalcula (x: Int, acumulador: Int) : Int =
            if ( x <= 1) acumulador
            else factCalcula (x - 1, x * acumulador)

        factCalcula(n ,1)
    }
}
```

# Recursividad

```
/* otroFactorial (10) = factCalcula (10,1)
= factCalcula (9, 10 * 1)
= factCalcula (8, 9 * 10 * 1)
= factCalcula (7, 9 * 9 * 10 * 1)
=.....
= factCalcula (2, 3 * 4 * ...* 10 * 1)
= factCalcula (1, 2 * 3 * 4 *...* 10 * 1)
= 1 * 2 * 3 * 4 * ... * 10
*/
```

# Recursividad

## Ejercicios

1. Función que concatena una cadena n veces

# Strings

```
val str: String = "Hola, yo estoy aprendiendo Scala"

//1. charAt --> accede al carácter en el índice 2 de la cadena
println(str.charAt(2))

//2. substring --> devuelve los caracteres de la posición 7 hasta 11
println(str.substring(7,11))

//3. split --> se utiliza para dividir la cadena en partes.
//toList convierte un array resultante del split en una lista de Scala
println(str.split(" ").toList)
```

# Strings

```
//4. startsWith --> verifica si el string comienza por el valor indicado. Devuelve un booleano  
println(str.startsWith("Hola"))  
  
//5.replace --> reemplaza carácter por otro  
println(str.replace(" ","-"))  
  
//6. toLowerCase --> convierte todos los caracteres de la cadena en minúsculas  
println(str.toLowerCase())  
  
//7. length --> devuelve la longitud  
println(str.length)
```

# Strings

```
//8. +: operador para añadir al inicio de la cadena. :+ operador para añadir al final de la cadena
val aNumberString="2"
val aNumber = aNumberString.toInt

println('a' +: aNumberString :+ 'z')

//9. imprime al revés la cadena
println(str.reverse)

//10. take (2) --> devuelve los dos primeros caracteres de la cadena
println(str.take(2))
```

# Strings

```
//11. Scala-specific: String interpolators ---s-interpolators
//Esto permite insertar valores de variables directamente en el texto utilizando el símbolo $.
val name = "David"
val edad = 12
val presenta = s"Hola, mi nombre es $name y tengo $edad años"
val otropresenta =s"Hola, mi nombre es $name y cumpliré ${edad + 1} años"
println(otropresenta)
```

# Strings

```
//12. F-interpolators
//El prefijo f permite especificar el formato para los valores interpolados dentro de la cadena.
//$nombre%s --> %s: Indica que el valor de la variable nombre debe tratarse como una cadena de texto (string)
//$velocidad%2.2f -->
//    %f: Indica que el valor de velocidad es un número en punto flotante (float).
//    2.2:
//        El primer 2 especifica que el número tendrá un ancho mínimo de 2 caracteres
//        (rellenará espacios si el número es más corto).
//    El segundo 2 especifica que tendrá 2 dígitos después del punto decimal.

val velocidad = 1.2f
val nombre = "Roberto"
println (f"$nombre%s puede comer $velocidad%2.2f bocadillos por minuto")
```

# Strings

```
//13. raw-interpolator --> los caracteres especiales se imprimen literalmente, sin efectos especiales
println("Esto es una \n nuevalinea")
println(raw"Esto es una \n nuevalinea")
val linea = "Esto es una \n nuevalinea"
println(raw"$linea")
```

# Ejercicio

```
/* Ejercicio: Calcula Descuento:  
Realiza una función que reciba los siguientes parámetros:  
    - precio del producto  
    - porcentaje de descuento  
La función debe calcular el precio final después de aplicar el descuento.  
Si el descuento es mayor o igual a 50, debe devolver una advertencia:  
    "Precio final: [preciofinal]. ¡Descuento muy alto!"  
Si el descuento es menor a 50, simplemente devuelve:  
    "Precio final: [preciofinal]"  
  
Una vez implementada la función realiza las siguientes llamadas:  
    - Calcula Descuento precio de 100  
    - Calcula Descuento precio de 200 y descuento de 20  
    - Calcula Descuento precio de 300 y descuento de 50  
*/
```

# Ejercicio 1: Calcular si un número es divisible por otro

```
/* Ejercicio 1 - Calcular si un número es divisible por otro  
En esta función el primer parámetros se pasa por llamada por valor y el segundo parámetro (divisor)  
se pasa por llamada por nombre.  
- Si el divisor es igual 0 entonces muestra el siguiente mensaje:  
  "El divisor no puede ser 0"  
- Si el divisor es mayor 0 entonces muestra el siguiente mensaje:  
  "Calculando si número es divisible por el divisor"  
  **número es el valor del parámetro y divisor es el valor del parámetro  
*/
```

## Ejercicio 2: Calcular el total de un pedido con impuestos y envío

```
/*Ejercicio 2 - Calcular el total de un pedido con impuestos y envío
```

La función que calcula el total de un pedido en línea, donde la función recibe como parámetros:

- Subtotal: Valor total de los productos
- Tasa de impuesto: El porcentaje de impuestos a pagar
- Envío: El coste de envío a pagar

Una vez realizado el cálculo la función debe devolver un mensaje que muestre la siguiente información con los valores formateados con dos decimales.

- Precio total
- Subtotal
- Tasa de impuestos
- Envío

```
*/
```

# Ejercicio 3: Buscar un elemento en una lista

```
/*Ejercicio 3 - Buscar un elemento en una lista
Implementa una función que busque si el número existe en la lista de números,
devolviendo verdadero o falso según sea el caso.
- Parámetros de la función serán los siguientes:
    List [Int] --> lista de enteros
    numero de tipo Int

    *** Pista:
        Lista.head --> obtenemos el primer elemento de la lista
        Lista.tail --> es una nueva lista que contiene todos los elementos de la lista original excepto el primero
*/
```

# Ejercicio 4: Invertir una lista de enteros

```
/*Ejercicio 4 - Inversión de una lista
```

Escribe una función recursiva que reciba una lista y devuelva la misma lista pero invertida.

Parámetros:

- Lista de enteros. Lista[Int]

\*\*\* Pista:

- Nil --> es una lista vacía y es una construcción esencial para trabajar en Scala.
- Se utiliza para terminar la recursión de una lista y sirve como marcador de final de una lista
- lista.last --> obtener el último elemento de la lista o lanza una excepción si la lista está vacía.
- lista.init --> para obtener todos los elementos de una lista excepto el último.

```
*/
```

# Ejercicio 5: Contar la cantidad de veces que aparece una carácter en una cadena

```
/* Ejercicio 5 - Contar la cantidad de veces que aparece un carácter en una cadena
Escribe una función recursiva que reciba una cadena de texto y un carácter y devuelva el número de veces que el
carácter aparece en la cadena.
Parámetros de entrada:
    | - Cadena de texto String y un carácter Char.
*/

```

# Object Oriented Basics

```
//constructor
✓ class Persona (nombre: String, val edad: Int) {
    //cuerpo
    val x = 2 //son campos
✓ println (1 + 3) //expresiones
//método --//overloading
def saluda(nombre: String): Unit = println(s"${this.nombre} dice: Hola, $nombre")
def saluda(): Unit = println (s"Hola, Yo soy $nombre")
✓ //def saluda(): Int = 43 --> En este caso el compilado sí se confunde

//multiple constructores
def this(nombre: String) = this(nombre,0)
def this() = this ("Juan Carlos")
}
```

# Constructor

```
//constructor  
class Persona (nombre: String, val edad: Int) {  
    ...}
```

Se declara una clase Persona con dos parámetros:

- Nombre de tipo String: parámetro privado sólo accesible desde la clase
- Edad de tipo entero: parámetro público accesible desde fuera de la clase por “val”.

# Cuerpo

```
//cuerpo  
val x = 2 //son campos  
println (1 + 3) //expresiones
```

val x = 2 → se define una variable inmutable con el valor 2 como propiedad de la clase.

Println (1 + 3) → se define una expresión que imprime el valor 4.

# Cuerpo : Sobrecarga

```
//método --//overloading
def saluda(nombre: String): Unit = println(s"${this.nombre} dice: Hola, $nombre")
def saluda(): Unit = println (s"Hola, Yo soy $nombre")
def saluda(): Int = 43 --> En este caso el compilado sí se confunde
```

Las dos primeras funciones se denominan igual pero con diferencias:

**Def saluda (nombre:String)**...→ declara un argumento y devuelve una interpolación de cadenas para incluir variables de texto. \${this.nombre} accede al atributo privado nombre de la clase.

**Def saluda ()** que no requiere argumentos y utiliza el nombre del objeto.

**De saluda ()** que devuelve un entero el compilador no sabe distinguir entre la segunda función y la tercera función.

# Cuerpo : Múltiples constructores

```
//multiple constructores  
def this(nombre: String) = this(nombre,0)  
def this() = this ("Juan Carlos")
```

**def this (nombre: String) = this(nombre, 0)** → permite crear el objeto sólo con el nombre, asignando la edad por defecto como 0.  
**def this() = this ("Juan Carlos")** → permite crear un objeto sin argumentos, asignando Juan Carlos como nombre y 0 como edad

# Instancia de la clase Persona

```
val persona = new Persona ("Carlos", 25)
println(persona.edad)
println(persona.x)
persona.saluda("Daniel")
persona.saluda()
```

**Val persona = new Persona("Carlos", 25)** → objeto de la clase Persona con el nombre Carlos y edad 25. Y con val declara una variable inmutable que almacena la instancia.

**Println (persona.edad)** → accede a la propiedad pública edad que se declaró con val en el constructor.

# ¿Qué imprime el método saluda?

```
val persona = new Persona ("Carlos", 25)
println(persona.edad)
println(persona.x)
persona.saluda("Daniel")
persona.saluda()
```

```
def saluda(nombre: String): Unit = println(s"${this.nombre} dice: Hola, $nombre"
def saluda(): Unit = println (s"Hello, Yo soy $nombre")
```

Persona.saluda ("Daniel") → ¿Qué imprime?

Persona.saluda() → ¿Qué imprime?

# Resultado de la impresión del método saluda

```
val persona = new Persona ("Carlos", 25)
println(persona.edad)
println(persona.x)
persona.saluda("Daniel")
persona.saluda()
```

```
def saluda(nombre: String): Unit = println(s"${this.nombre} dice: Hola, $nombre"
def saluda(): Unit = println (s"Hello, Yo soy $nombre")
```

Persona.saluda ("Daniel") → Carlos dice: Hola, Daniel  
Persona.saluda() → Hola, Yo soy Carlos.

# Resumen

```
class Persona(nombre: String, edad:Int) //Definición de clases
val persona = new Persona("Pedro", 25) //Instancia
class Persona(val nombre: String, edad: Int) //Parámetros vs Campos

def saluda(): String = {....} //Definición de métodos
val personaSaluda = persona.saluda //Llamada a métodos

//sintaxis para métodos sin parámetros
//la palabra clave this
```

# Más ejemplos

```
/* Novela y Escritor
   Escritor: nombre, primer apellido, año
   - método nombrecompleto

   Novela: nombre, año de realización, autor
   - Edad del autor
   - EscritoPor (autor)

*/
```

## Más ejemplos

```
class Escritor (nombre: String, primerApellido: String, val año: Int){  
    def nombreCompleto: String = nombre + " " + primerApellido  
}
```

### Clase Escritor:

- nombre y primerApellido: strings privados
- año: un valor público (usando val)

# Más ejemplos

```
class Novela (nombre: String, año: Int, autor: Escritor) {  
    def edadAutor = año - autor.año  
    def EscritoPor(autor:Escritor) = autor == this.autor  
}
```

Clase Novela:

- nombre: el título de la novela
- año: año de publicación
- autor: instancia de la clase escritor

EdadAutor: calcula la edad que tenía el autor cuando escribió la novela

EscritoPor: devuelve verdadero si el autor es el mismo que escribió la novela

# Más ejemplos: ¿Qué imprime por pantalla?

```
val autor = new Escritor("Carlos", "Herranz", 1920)
val novela = new Novela ("Gran Novela", 1930, autor)

println(novela.edadAutor)
> println(novela.EscritoPor(autor))
```

```
class Novela (nombre: String, año: Int, autor: Escritor) {
    def edadAutor = año - autor.año
    def EscritoPor(autor:Escritor) = autor == this.autor
}
```

# Ejercicio

```
/* Clase contador:  
    Parámetros:  
        - recibe un valor entero por defecto 0  
    Funciones:  
        - función para incrementar + 1 el contador  
        - función para decrementar - 1 el contador  
        - función de imprimir por pantalla el contador  
*/
```

# Method Notations

```
class Persona (val nombre: String, peliculaFavorita: String){  
    def meGusta (pelicula: String): Boolean = pelicula == peliculaFavorita
```

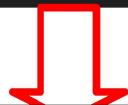


```
val maria = new Persona ("Maria","Los vengadores")  
//NOTACIONES INFIJOS = OPERADORES  
println(maria.meGusta("Los vengadores")) //True  
println(maria meGusta "Los vengadores") //True
```

1. Definir una clase Persona con dos parámetros nombre y peliculaFavorita.
2. Definir un método meGusta
3. Instanciar la clase persona
4. Imprimir por pantalla

# Method Notations

```
class Persona (val nombre: String, peliculaFavorita: String){  
    def meGusta (pelicula: String): Boolean = pelicula == peliculaFavorita  
    def salirCon (persona: Persona ): String = s"${this.nombre} sale con ${persona.nombre}"
```

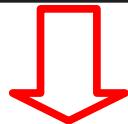


```
// operadores en Scala  
val pedro = new Persona ("Pedro", "Tiburón")  
println (maria.salirCon(pedro))  
println (maria salirCon pedro)
```

5. Definimos método salirCon dentro de la clase Persona
6. Nueva instancia de la clase
7. Imprime pantalla

# Method Notations

```
class Persona (val nombre: String, peliculaFavorita: String){  
    def meGusta (pelicula: String): Boolean = pelicula == peliculaFavorita  
    def salirCon (persona: Persona ): String = s"${this.nombre} sale con ${persona.nombre}"  
    def + (persona: Persona ): String = s"${this.nombre} no le gusta ${persona.nombre}" //método llamado + es válido
```



```
// operadores en Scala  
val pedro = new Persona ("Pedro", "Tiburón")  
println (maria + pedro)  
println (maria.+(pedro))
```

8. Nuevo método denominado +
9. Imprimir pantalla

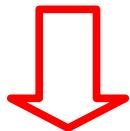
# Method Notations

## TODOS LOS OPERADORES SON MÉTODOS

```
//TODOS LOS OPERADORES SON MÉTODOS  
println(1 + 2)  
println (1.+ (2))
```

# Method Notations

```
def unary_! : String = s"$nombre, que es esto?" //debe haber un espacio entre unary_! y los dos puntos
```



```
//NOTACION PREFIJOS
println(!maria)
println(maria.unary_!)
```

# Method Notations

```
//NOTACION más PREFIJOS  
val x = -1 //equivalente con 1.unary_-  
val y = 1.unary_-  
val z = 2.unary_+  
val j = 3.unary_~  
//unary_prefix solo con - + ~ !  
  
println(y) // valor negativo  
println(z) //valor positivo  
println(j) //El operador ~ tiene sentido en operaciones bit a bit,  
//útil en manipulación de bits, como en programación de bajo nivel  
// o en el desarrollo de aplicaciones que requieren optimización binaria.  
//Este método se encuentra definido para tipos como Int y Long en Scala.
```

# Method Notations

```
class Persona (val nombre: String, peliculaFavorita: String){  
    def meGusta (pelicula: String): Boolean = pelicula == peliculaFavorita  
    def salirCon (persona: Persona ): String = s"${this.nombre} sale con ${persona.nombre}"  
    def + (persona: Persona ): String = s"${this.nombre} no le gusta ${persona.nombre}" //método llamado + es válido  
  
    def unary_! : String = s"$nombre, que es esto?" //debe haber un espacio entre unary_! y los dos puntos  
}
```

```
def apply(): String = s"Hola, mi nombre es $nombre y me gusta $peliculaFavorita"
```



```
//apply  
println(maria.apply())  
println(maria()) //equivalente
```

# Ejercicio 1: Mascota

```
/* Ejercicio 1: Mascota y su actividad favorita
Simula la creación y gestión de una mascota con un nombre y una actividad favorita.
Datos de la mascota:
- Nombre
- Actividad favorita
Operaciones:
- Cambiar la actividad favorita a Jugar con la pelota.
- Mostrar la información actualizada de la mascota.
Clase Mascota:
- Contiene los parámetros de entrada nombre y actividad favorita
- Métodos:
  - cambiarActividad(nuevaActividad: String): Cambia la actividad favorita de la mascota.
  - mostrarInfo(): Muestra los detalles de la mascota, incluyendo su nombre y actividad favorita.
*/
```

# Ejercicio 2: Tienda

Ejercicio TiendaApp:

Simula las operaciones de venta y reposición en una tienda.

Datos del producto:

- nombre: Camiseta
- precio: 25
- Cantidad en stock = 20

Operaciones:

- Vender 5 camisetas
- Agregar 10 camisetas al stock
- Mostrar información actualizada

Clase Producto:

- Contiene atributos para el nombre, precio y cantidad en stock.
- Métodos:
  - vender(cantidad: Int): Reduce el stock si hay suficiente cantidad disponible y la cantidad es mayor que 0.  
Muestra un mensaje de error por pantalla si no hay suficiente stock o si la cantidad es inválida.
  - agregarStock(cantidad: Int): Aumenta el stock si la cantidad es mayor que 0.  
Muestra un mensaje de error por pantalla si la cantidad es inválida.
  - mostrarInfo(): Imprime por pantalla los detalles del producto (Producto, precio y cantidad)

## Ejercicio 3: Cuenta Bancaria

```
/* Ejercicio 2: Cuenta Bancaria
Simula las operaciones básicas de una cuenta bancaria.

Datos de la cuenta:
    Número de cuenta: 987654321
    Saldo inicial: 1000

Operaciones:
    - Depositar 500
    - Retirar 200
    - Intentar retirar 1500 (¡cuidado excede el saldo!)
    - Mostrar información de la cuenta bancaria actualizada
```

# Ejercicio 3: Cuenta Bancaria

Clase CuentaBancaria:

- Contiene parámetros de entrada para el número de cuenta, saldo actual de la cuenta.
- Métodos:
  - depositar(cantidad: Double):
    - 1.- Incrementa el saldo si la cantidad es mayor que 0.  
Muestra el siguiente mensaje por pantalla con el total depositado y el saldo actual:  
"Se han depositado 500 en la cuenta 987654321. Saldo actual: 1500"
    - 2.- Muestra el siguiente mensaje por pantalla si la cantidad es menor de 0 o igual a 0:  
" No se puede depositar una cantidad negativa o cero"
  - \*\*\*Pista: saldo += cantidad
  - retirar(cantidad: Double): Disminuye el saldo si la cantidad es mayor que 0 y no excede el saldo disponible.
    - 1.- Si la cantidad es mayor que saldo muestra el siguiente mensaje por pantalla:  
"No hay suficiente saldo para retirar 1500. Saldo actual: 1300"
    - 2.- Si la cantidad es menor que saldo muestra el siguiente mensaje por pantalla:  
"Se han retirado 200 de la cuenta 987654321. Saldo actual: 1300"
    - 3.- Si la cantidad es negativa o 0 muestra el siguiente mensaje por pantalla:  
"No se puede retirar una cantidad negativa o cero"
  - \*\*\*Pista: saldo -= cantidad

# Objects

```
| object Persona { //tipo + una instancia única
|   val n_ojos = 2
|   def vuela: Boolean = false
| }
| println(Persona.n_ojos)
| println(Persona.vuela)
```

# Objects

```
object Persona { //tipo + una instancia única
    val n_ojos = 2
    def vuela: Boolean = false
}
println(Persona.n_ojos)
println(Persona.vuela)

//podemos tener objetos y clases con el mismo nombre.
// Este patrón se llama Companions o compañeros residen en el mismo ámbito
class Persona {
    //funcionalidad a nivel de instancia
}
```

# Objects

```
//Objeto Scala = Instancia Singleton
val maria = Persona
val pedro = Persona
println(maria == pedro) //maria y pedro apuntan a la misma instancia que es el objeto Persona --> true

//Clase
val cristina = new Persona
val roberto = new Persona
println(cristina == roberto) //cristina y roberto son instancias distintas de la clase Persona --> false
```

# Objects

```
object Persona { //tipo + una instancia única
    val n_ojos = 2
    def vuelo: Boolean = false
    //factory method --propósito construir personas
    def padres(madre: Persona, padre: Persona): Persona = new Persona("Carlos")
}


```

```
class Persona (val nombre: String) {
    //funcionalidad a nivel de instancia
}
```

# Objects

```
object Persona { //tipo + una instancia única
    val n_ojos = 2
    def vuelo: Boolean = false
    //factory method --propósito construir personas
    def padres(madre: Persona, padre: Persona): Persona = new Persona("Carlos")
}


```

```
class Persona (val nombre: String) {
    //funcionalidad a nivel de instancia
}
```

# Objects

```
//Clase
val cristina = new Persona ("Cristina")
val roberto = new Persona ("Roberto")
println(cristina == roberto) //cristina y roberto son instancias distintas de la clase Persona --> false

val carlos = Persona.padres(cristina, roberto)
```

# Objects

```
object Persona { //tipo + una instancia única
    val n_ojos = 2
    def vuelo: Boolean = false
    //factory method --propósito construir personas
    //def padres(madre: Persona, padre: Persona): Persona = new Persona("Carlos")
    def apply(madre: Persona, padre: Persona): Persona = new Persona("Carlos")
}

//val carlos = Persona.padres(cristina, roberto)
val carlos = Persona(cristina,pedro)
```

# Objects

## ¿Qué es un Singleton en Scala?

Un objeto declarado con object es una clase que solo puede tener una única instancia en todo el programa.

**Companion Object:** Un object puede estar emparejado con una clase del mismo nombre para compartir acceso a miembros privados de la clase. E

## ¿Para qué sirve Companion Object?

- Sirve para inicializar, validar o realizar cálculos estáticos. Por ejemplo, crear objetos preconfigurados o con valores predeterminados.
- Simular métodos estáticos. Scala no tiene métodos estáticos como en Java, el companion Object puede actuar como si fueran métodos estáticos.
- Patrón Fábrica (Factory Pattern): Permite crear instancias de una clase de forma más elegante y controlada, ocultando la lógica del constructor.
- Compartir constantes o configuraciones comunes a todas las instancias de la clase.
- Simplificar la sintaxis con “apply”

# Ejemplo

```
object Object_tercera extends App {  
    //Clase calculadora con dos métodos sumar y restar dos números.  
    //Crear instancias de la calculadora con un valor inicial  
    //Proporcionar un método estático para sumar dos números directamente  
    object Calculadora{  
        def iniciaCalc(valor: Int): Calculadora = new Calculadora (valor)  
        def sumarDirecta(a: Int, b: Int): Int = a + b  
    }  
    class Calculadora (val valor: Int) {  
        def suma(x: Int): Int = valor + x  
        def resta(x: Int): Int = valor - x  
    }  
    val calc = Calculadora.iniciaCalc(10)  
    println(s"Valor inicial ${calc.valor}")  
    val suma = calc.suma(5)  
    println(s"Resultado de sumar 5: $suma")  
    val sumaDirecta = Calculadora.sumarDirecta(3,7)  
    println(s"Resultado de sumar 3 y 7: $sumaDirecta")  
}
```

# Ejercicio

```
/*Crea una clase Libro que tenga un título y un autor.  
Usa el companion object para:  
    - Crear un libro con un título y autor específicos  
    - Crear un libro anónimo con título "Desconocido" y autor "Anónimo"  
  
Ejercicio:  
    1 - Crear un libro con un título "1984" y el autor "George Orwell"  
    2 - Crear un libro anónimo  
    3. Imprimir por pantalla los detalles de ambos libros  
*/
```

# Herencia

```
object Herencia extends App {  
  
    class Animal {  
        def comer = println("comer comer")  
    }  
  
    class Gato extends Animal  
  
    val gato = new Gato //instancia de la clase Cat  
    gato.comer //devuelve comer comer  
}
```

PROBLEMS

12

OUTPUT

DEBUG CONSOLE

TERMINAL

PORTS

comer comer

# Herencia: Métodos Private

```
object Herencia extends App {  
  
    class Animal {  
        private def comer = println("comer comer")  
    }  
  
    class Gato extends Animal  
  
    val gato = new Gato //instancia de la clase Cat  
    gato.comer  
}
```

Cuando un método o una variable se declara como **private**, su visibilidad está restringida a la clase en la que se define.

# Herencia: Métodos Protected

```
object Herencia extends App {  
  
    class Animal {  
        protected def comer = println("comer comer")  
    }  
  
    class Gato extends Animal {  
        def comerGato = {  
            comer //el método protegido sí se será accesible desde la subclase, pero no es accesible fuera de la clase  
            println("comida gato")  
        }  
    }  
  
    val gato = new Gato //instancia de la clase Cat  
    //gato.comer  
    gato.comerGato  
}
```

Cuando método o una variable se declara como **protected**, su visibilidad está limitada a la clase en la que se declara o las subclases que heredan esta clase.

# Herencia

Acceso	Visibilidad	Ejemplo
<b>private</b>	Sólo dentro de la clase Animal	private def comer= println("comer comer")
<b>protected</b>	Dentro de Animal y su subclases (Gato)	protected def comer= println("comer comer")
<b>Por defecto</b>	Público accesible	def comer =println("comer comer")

# Herencia

```
//constructores
class Persona (nombre: String, edad: Int)
class Adulto (nombre: String, edad: Int, carnetConducir: String) extends Persona
```

¿Por qué da error?

# Herencia

```
//constructores  
class Persona (nombre: String, edad: Int)  
class Adulto (nombre: String, edad: Int, carnetConducir: String) extends Persona (nombre, edad)
```

Porque lo primero que se realiza es crear la instancia de la clase Persona que necesita dos parámetros nombre y edad.

# Herencia: Override

```
//overriding --anulando
class Perro extends Animal {
    ↑ comer
    | override def comer = println("comer perro")
}
val perro = new Perro
perro.comer
```

**Override** se utiliza para anular un método con el mismo nombre de la clase padre.

La subclase puede necesitar implementar el método reemplazando el comportamiento definido en la clase padre.

# Herencia: Override

```
class Animal {  
    val TipoCriatura = "Salvaje"  
    protected def comer = println("comer comer")  
}
```

```
//overriding --anulando  
class Perro extends Animal {  
    ↑ TipoCriatura  
    override val TipoCriatura = "Domestica"  
    ↑ comer  
    override def comer = println("comer perro")  
}
```

```
println(perro.TipoCriatura)
```



¿Qué imprime por pantalla?

# Herencia: Override

```
↑ TipoCriatura
class Perro (override val TipoCriatura: String) extends Animal {
    ↑ comer
    | override def comer = println("comer perro")
}

val perro = new Perro ("Indiferente")
println(perro.TipoCriatura)
```

# Herencia: Polimorfismo de subtipos

```
//tipo sustitución: polimorfismo
class Animal {
    val TipoCriatura = "Salvaje"
    def comer = println("comer comer")
}

↑ TipoCriatura
class Perro (override val TipoCriatura: String) extends Animal {
    ↑ comer
    override def comer = println("comer perro")
}
val esAnimal: Animal = new Perro ("Ni idea")
esAnimal.comer
```

El polimorfismo de subtipos se basa en la herencia y permite trabajar con tipos derivados a través de una referencia al tipo base

¿Qué imprime por pantalla esAnimal.comer?

# Herencia: Super

```
//super

class Animal {
    val TipoCriatura = "Salvaje"
    def comer = println("comer comer")
}
↑ TipoCriatura
class Perro2 (override val TipoCriatura: String) extends Animal {
    ↑ comer
    override def comer ={
        super.comer
        println ("comer perro 2")
    }
}
val perro2 = new Perro2 ("domestica")
perro2.comer
```

Esto asegura que el comportamiento original del método de la clase base se ejecute antes (o después, dependiendo del orden) de añadir el nuevo comportamiento en la clase derivada.

# Herencia: Impedir la anulación

```
//impedir la anulación
//1- utilizar la palabra reservada final
class Animal {
    val TipoCriatura = "Salvaje"
    final def comer = println("comer comer")
}

class Perro (override val TipoCriatura: String) extends Animal {
    override def comer = {
        super.comer
        println("comer Perro")
    }
}
```

**ERROR PORQUE  
NO PERMITE  
QUE SEA  
SOBRESCRITO**

# Herencia: Impedir la anulación

```
//2- utilizar final en toda la clase
final class Animal {
    val TipoCriatura = "Salvaje"
    def comer = println("comer comer")
}

class Perro (override val TipoCriatura: String) extends Animal {
    override def comer = {
        super.comer
        println("comer Perro")
    }
}
```

**FINAL:**  
impide que una clase  
o un miembro  
(método o campo)  
sea sobrescrito o  
extendido.

# Herencia

```
//3- sellar la clase = extender clases en este fichero, para prevenir la extensión en otros ficheros
sealed class Animal {
    val tipocriatura = "Salvaje"
    def comer = println ("comer comer")
}
```

**El compilador sabe todas las posibles subclases de una clase sealed porque están en el mismo archivo**

```
↑ Tipocriatura
class Perro (override val Tipocriatura: String) extends Animal {
    ↑ comer
    override def comer = {
        super.comer
        println("comer Perro")
    }
}
```

# Ejercicio

/\*

Contexto:

Se necesita modelar distintos tipos de vehículos utilizando herencia en Scala.

La clase base debe definir un comportamiento y un atributo predeterminados, mientras que las clases hijas sobrescribirán estos elementos para agregar comportamientos específicos.

Clase Vehiculo

Crea una clase sellada llamada Vehiculo para restringir su extensión a un solo archivo.

Atributo: val TipoVehiculo con valor predeterminado "General".

Método de conducir, que imprima "Conduciendo vehículo".

Clase Hija: Coche

Crea una clase Coche que extienda la clase Vehiculo.

Sobscribe el atributo TipoVehiculo para que sea personalizado al crear una instancia.

Sobscribe el método conducir de la clase padre. El nuevo método debe:

Llamar al método original conducir de la clase padre utilizando super.conducir.

Imprimir "Conduciendo coche".

Llamadas:

Crea una instancia de la clase Coche indicando el tipo de vehículo (por ejemplo, "Sedán").

Imprime el valor de TipoVehiculo de la instancia.

Llama al método conducir para verificar el comportamiento sobrescrito.

\*/

# Tipos de datos Abstractos y clases

```
//abstract  
abstract class Animal {  
    val TipoCriatura : String  
    def comer: Unit  
}  
  
val animal = new Animal
```

Una clase abstracta no puede ser instanciada porque contiene atributos y /o métodos que no están implementados.

Para crear un objeto de una clase abstracta se debe primero crear una clase hija que sobreescriba e implemente todos los métodos y atributos.

Una clase abstracta sirve como una plantilla o modelo base que otras clases pueden heredar e implementar. Por ejemplo, puede ser útil definir una estructura común para un conjunto de clases relacionadas pero necesitas que cada clase hija implemente métodos o atributos.

# Tipos de datos Abstractos y clases

```
//abstract
abstract class Animal {
    val TipoCriatura : String
    def comer: Unit
}

class Perro extends Animal {
    ↑ TipoCriatura
    override val TipoCriatura: String = "Doméstico"
    ↑ comer
    def comer: Unit = println ("comer perro")
}
```

# Rasgos (Traits)

```
//abstract  
abstract class Animal {  
    val TipoCriatura : String  
    def comer: Unit  
}
```

1

```
class Perro extends Animal {  
    ↑ TipoCriatura  
    override val TipoCriatura: String = "Perro"  
    ↑ comer  
    def comer: Unit = println ("comer perro")  
}
```

2

# Rasgos (Traits)

```
//traits (rasgos)
trait Carnivoro {
  def comer (animal: Animal): Unit
}
```

3

```
class Cocodrilo extends Animal with Carnivoro {
  ↑ TipoCriatura
  val TipoCriatura: String = "croc"
  ↑ comer
  def comer: Unit = println("ñam ñam")
  ↑ comer
  def comer(animal: Animal): Unit =
    | println( s"Yo sou un cocodrilo y me como ${animal.TipoCriatura}")
```

4

# Rasgos (Traits)

```
val perro = new Perro  
val croc = new Cocodrilo  
croc.comer (perro)
```

5

## Características:

- 1.- Pueden contener métodos abstractos y concretos
- 2.- No pueden tener parámetros en el constructor
- 3.- Una clase puede heredar de la clase padre y de un trait o más con la palabra with
- 4.- Pueden combinar múltiples traits y redefinir si hay conflictos

## Uso:

- Cuando quieras redefinir comportamientos comunes que puedan ser reutilizados en múltiples clases.
- Herencia múltiple de forma ordenada y controlada

# Rasgos (Traits)

```
trait SangreFria
class Cocodrilo extends Animal with Carnivoro with SangreFria {
    ↑ TipoCriatura
    val TipoCriatura: String = "croc"
    ↑ comer
    def comer: Unit = println("ñam ñam")
    ↑ comer
    def comer(animal: Animal): Unit =
        |   | println(s"Yo sou un cocodrilo y me como ${animal.TipoCriatura}")
    }
val perro = new Perro
val croc = new Cocodrilo
croc.comer (perro)
```

# Rasgos (Traits) vs Clases Abstractas

```
trait Carnívoro (name: String) {  
    def comer (animal: Animal): Unit  
    val prefiereComida: String ="Carne fresca"  
}
```

¿Es correcta esta implementación?

# Rasgos (Traits) vs Clases Abstractas

```
// 1- rasgos (traits) no tienen parámetros constructores
/* Ejemplo: no permite parámetros */
trait Carnivoro (name: String) {
    def comer (animal: Animal): Unit
    val prefiereComida: String = "Carne fresca"
}
```

**ERROR TRAITS  
NO PUEDEN  
TENER  
PARÁMETROS**

```
class Cocodrilo extends Animal with Carnivoro with SangreFria {
    ↑ TipoCriatura
    override val TipoCriatura: String = "croc"
    ↑ comer
    def comer: Unit = println("ñam ñam")
    ↑ comer
    def comer(animal: Animal): Unit =
        | println(s"Yo sou un cocodrilo y me como ${animal.TipoCriatura}")
    }
```

# Rasgos (Traits) vs Clases Abstractas

	<b>Clase abstracta</b>	<b>Trait</b>
<b>Declaración</b>	- abstract class - puede tener constructores con parámetros	- trait - no puede tener parámetros constructores
<b>Herencia múltiple</b>	Solo puede heredar de una única clase abstracta (herencia simple)	Una clase puede mezclar múltiples traits usando extends y with. Herencia múltiple
<b>Parámetros</b>	Puede tener parámetros en el constructor y campos inicializados	No puede tener parámetros en el constructor Sí puede contener campos, pero deben inicializarse directamente o ser abstractos
<b>Rendimiento</b>	Más eficiente	Ligera sobrecarga
<b>Uso</b>	Jerarquía de clases, clases hijas comparten un comportamiento común. Es adecuada si necesitas constructores con parámetros	Comportamientos comunes o módulos reutilizables que pueden mezclar con múltiples clases. Herencia múltiple

# Ejemplo

```
abstract class Animal(nombre: String) {  
    def sonido(): String  
}  
class Perro(nombre: String) extends Animal(nombre) {  
    def sonido():String = "Guau"  
}  
  
trait Volador {  
    def volar(): String = "Estoy volando"  
}  
trait Nadador {  
    def nadar(): String = "Estoy nadando"  
}  
class Pato extends Volador with Nadador
```

# Ejercicio 1: Dispositivos electrónicos

```
/*Ejercicio 1. Dispositivos electrónicos
El sistema de Dispositivos electrónicos debe realizar las siguientes acciones:
- Encender: imprime por pantalla "El dispositivo está encendido"
- Apagar: imprime por pantalla "El dispositivo está apagado"

El dispositivo Teléfono debe realizar la acción de Llamar a un número
imprimiendo por pantalla "Llamando al número [número]"
El dispositivo Tablet deber realizar la acción de navegar Web
imprimiendo por pantalla "Navengando a [url]"
*/
```

## Ejercicio 2: Formas de pago

```
/*Ejercicio 2. Formas de pago
Sistema de Formas de pago deber cumplir que:
- Todas las formas de pago deben procesar el pago pasándole una cantidad.
- La forma de pago Tarjeta de Crédito debe imprimir por pantalla:
    "Procesando pago de [cantidad] con tarjeta de crédito"
- La forma de pago PayPal debe imprimir por pantalla:
    "Procesando pago de [cantidad] con PayPal"
- La forma de pago Transferencia Bancaria debe imprimir por pantalla:
    "Procesnado pago de [cantidad] mediante transferencia bancaria"
*/
```

# Ejercicio 3: Electrodomésticos Inteligentes

```
/*Ejercicio 3: Electrodomésticos Inteligentes
El sistema de Electrodomésticos Inteligentes debe cumplir que:
- Algunos electrodomésticos tienen la función de conexión WiFi que imprime por pantalla
  "Conectando el electrodoméstico a la red WiFi"
- Algunos otros tienen la función de programación automática donde se les pasa el tiempo e
  imprime por pantalla "Programando electrodoméstico a las [tiempo]"

El electrodoméstico Lavadora puede conectarse a WiFi y ser programada.
El electrodoméstico Refrigerador puede conectarse a WiFi.
*/
```

# Ejercicio 4: Robots Multifuncionales

```
/*Ejercicio 4: Robots Multifuncionales
Sistema para modelar diferentes tipos de robots que contienen las características de:
- Modelo que devuelve el modelo del robot, por ejemplo, "ModeloX".
- Batería que devuelve el nivel de batería del robot, por ejemplo, 80.
- Estado que devuelve el mensaje por pantalla de "El robot [modelo] tiene [bateria]% de batería"

Algunos de los modelos de robots pueden levantar peso imprimiendo por pantalla:
    "El robot está levantando [peso] kg."
Algunos de los modelos de robots puede asistir a personas imprimiendo por pantalla:
    "El robot está asistiendo a [persona]."

Existe dos tipos de robot:
- Robot que se denomina Obrero que realiza los trabajos pesados.
- Robot que se denomina Asistente que realiza la asistencia a personas
*/
```

# case class.

```
// Definición de una case class
case class Persona(nombre: String, edad: Int)

// Uso de la case class
object EjemploCaseClass extends App {
    // Crear una instancia
    val persona1 = Persona("Juan", 25)
    // Acceder a los campos
    println(s"Nombre: ${persona1.nombre}, Edad: ${persona1.edad}")
    // Crear una copia modificando uno de los campos
    val persona2 = persona1.copy(edad = 30)
    println(s"Nombre: ${persona2.nombre}, Edad: ${persona2.edad}")
    // Comparación (case classes implementan `equals` y `hashCode`)
    println(persona1 == persona2) // false
    // Representación como cadena (toString implementado automáticamente)
    println(persona1) // Persona(Juan,25)
}
```

# Case Classes. Propiedades. Diferencias con class

## Inmutabilidad

case class	class
Las propiedades declaradas como parámetros en una case class son inmutables (implícitamente val)	Las propiedades deben declararse explícitamente como val o var para definir su mutabilidad.

# Case Classes. Propiedades. Diferencias con class

## Creación de Instancias

case class	class
No requiere la palabra clave <i>new</i> .	Requiere la palabra clave <i>new</i> .

# Case Classes. Propiedades. Diferencias con class

## Métodos generados

Case class	class
Genera automáticamente métodos como <code>ToString</code> , <code>equals</code> , <code>copy</code>	No genera estos métodos automáticamente. Se deben implementar manualmente si son necesarios.

# Case Classes. Propiedades. Diferencias con class

## Pattern Matching o Coincidencia de patrones

case class	class
Es compatible con el pattern matching de forma predeterminada	No es compatible con el pattern matching a menos que se implemente manualmente el método <i>(método para extraer información de los objetos)</i>

El Pattern Matching evalúa una expresión y la compara con una serie de patrones predefinidos. Cuando encuentra un patrón que coincide, ejecuta el bloque de código asociado a ese patrón.

# Case Classes. Propiedades. Diferencias con class

## Pattern Matching o Coincidencia de patrones

// Usar pattern matching para determinar el mensaje basado en la edad

```
persona match {  
    case Persona(_, edad) if edad < 18 => println("Es un menor de edad.")  
    case Persona(_, edad) if edad >= 18 && edad <= 65 => println("Es un adulto.")  
    case Persona(_, edad) if edad > 65 => println("Es una persona mayor.")  
}
```

\* case Persona (\_, edad) => el \_ indica que se está ignorando el campo nombre de la instancia Persona. Es decir, no nos interesa el valor del nombre en este patrón, solo nos interesa el valor de edad

# Case Classes. Propiedades. Diferencias con class

## Comparación

Case class	class
La comparación es estructural. Compara los valores de los campos.	La comparación es referencial por defecto. Compara referencias de memoria

# Case Classes. Propiedades. Diferencias con class

## Clonado

Case class	class
Proporciona el método <i>copy</i> para crear copias con valores modificados	No tiene un método <i>copy</i> . Se debe implementar manualmente

# Case Classes. Propiedades. Diferencias con class

## Serialización

Case class	class
Las case class son serializables por defecto	No son serializables automáticamente. Se debe implementar manualmente

La serialización es el proceso de convertir un objeto en una representación que pueda ser almacenada o transmitida. Ejemplos, guardar un objeto en un archivo; enviar un objeto a través de la red; almacenar un objeto en caché.

# Case Classes. Propiedades. Diferencias con class

## Herencia

Case class	class
No puede extender directamente otra case class pero puede extender una clase o trait.	Puede heredar de cualquier clase o trait.

# Case Classes. Propiedades. Diferencias con class

## Uso de colecciones

Case class	class
Ideal para usar como claves en mapas (Map) o en estructuras que requieran hashcodes confiables.	No recomendado para este uso sin sobreescribir explícitamente equals y hashCode.

Estructuras hashCode es un valor numérico generado a partir de un objeto utilizando una función hash. Este código representa de manera única el contenido del objeto. Ejemplo:  
val str = "Scala"  
Println (str.hashCode()) → resultado: 83835195

# ¿Cuándo usar case class?

- 1. Necesitas comparaciones por valor (no por referencia)**
- 2. Necesitas las instancias inmutables**
- 3. Quieres desestructuración fácil (Pattern Matching)**
- 4. Necesitas crear instancias de manera concisa**
- 5. Generación automática de `toString`, `copy`, `hashCode`**

# Case Classes

## 1. Los parámetros de la clase son campos

```
case class Persona (nombre: String, edad: Int)
//1. Los parámetros de la clase son campos
val Roberto = new Persona ("Roberto", 25)
println (Roberto.nombre)
```

# Case Classes

## 2. sensible `toString`

```
//2. sensible toString  
println(Roberto.toString)
```

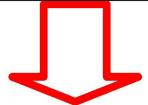


```
Persona(Roberto, 25)
```

# Case Classes

## 3. equals and hashCode

```
//3. equals and hashCode  
val Roberto2 = new Persona ("Roberto", 25)  
println (Roberto == Roberto2)
```

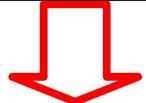


¿Verdadero o Falso?

# Case Classes

## 3. equals and hashCode

```
//3. equals and hashCode  
val Roberto2 = new Persona ("Roberto", 25)  
println (Roberto == Roberto2)
```



Verdadero (True)

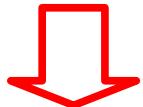
# Case Classes

## 4. Case Classes se puede copiar

```
//4. CCs se puede copiar métodos  
val Roberto3 = Roberto.copy(edad = 45)  
println (Roberto3)
```



En este caso Roberto3, tiene el mismo nombre pero la edad es distinta



```
Persona(Roberto,45)
```

# Case Classes. Ejercicio 1

```
/*Ejercicio 1: Utiliza case class  
Un libro tiene las siguientes propiedades:  
- Título: Nombre del libro de tipo cadena.  
- Autor: Nombre del autor de tipo cadena.  
- Año de publicación: El año de tipo entero.
```

Se debe realizar las siguientes acciones:

- Crea un libro con los siguientes valores:
  - Título: "1984"
  - Autor: Anónimo
  - Año: 1986
- Imprimir por pantalla los datos del libro

```
*/
```

## Case Classes. Ejercicio 2

```
/*Ejercicio 2
Representar diferentes tipos de operaciones matemáticas: Suma, Resta y Multiplicación.
Cada operación tendrá dos operandos (a y b) y utilizamos pattern matching para realizar
la operación correspondiente y devolver el resultado
*/
```

# Enums

```
object Enums {
    enum Permisos {
        case READ, WRITE, EXECUTE, NONE
        //añadir métodos/campos
        def abrirDocumento(): Unit =
            if (this == READ) println("abrir documento...")
            else println ("Lectura no permitida")
    }
    val algunosPermisos: Permisos = Permisos.READ
    def main (args: Array[String]): Unit = {
        algunosPermisos.abrirDocumento()
    }
}
```

# Enums. Ejercicio

```
/* Ejercicio 1: Colores primarios
 - Representa los colores primarios: Rojo, azul y amarillo.
 - Imprime por pantalla todos los colores primarios
 */
// Definimos un enum para representar los colores primarios
```

# Exceptions

```
val x:String = null  
println (x.length)
```

```
Exception in thread "main" java.lang.ExceptionInInitializerError  
at Exceptions.main(13_Exceptions.scala)
```

```
Caused by: java.lang.NullPointerException: Cannot invoke "String.length()" because the return value of "Exceptions$.x()" is null  
at Exceptions$.<clinit>(13_Exceptions.scala:4) Exceptions.scala:1
```

# Exceptions

```
//Excepciones
def obtenerEntero(conException: Boolean): Int =
    if (conException) throw new RuntimeException("No es un entero")
    else 24

try {
    //código que debe ser capturada la excepción
    obtenerEntero (true)
} catch {
    case e: RuntimeException => println ("capturado Runtime exception")
} finally {
    //código que se ejecutará
    println("finally")
}
```

# Exceptions: ¿Qué pasa?

```
//Excepciones
def obtenerEntero(conException: Boolean): Int =
    if (conException) throw new RuntimeException("No es un entero")
    else 24

try {
    //código que debe ser capturada la excepción
    obtenerEntero (true)
} catch {
    case e: NullPointerException => println ("capturado Runtime exception")
} finally {
    //código que se ejecutará
    println("finally")
}
```

# Exceptions. Define tus propias excepciones

```
val falloPotencial = try {  
    //código que debe ser capturada la excepción  
    obtenerE+...  
} catch { type RuntimeException: RuntimeException  
    case e: RuntimeException => println ("capturado Runtime exception")  
} finally {  
    //código que se ejecutará  
    // es opcional  
    // no influye en el resultado de la expresión  
    //usa finally sólo si tiene algún efecto  
    println("finally")  
}  
  
println(falloPotencial)
```

# Exceptions. Ejercicio 1

```
/* Ejercicios
1. Calculadora
    - suma (x,y)
    - restar (x,y)
    - multiplicar (x,y)
    - dividir (x,y)

    Throw
        - OverflowException si suma (x,y) excede int.MAX_VALUE
        - UnderflowException si la resta (x,y) excede int.MIN_VALUE
        - MathCalculationException si la división por 0
*/
```

## Exceptions. Ejercicio 2

```
/*Ejercicio 2: Manejo de Excepciones con Conversión de tipos  
Función para convertir una cadena a un tipo entero.  
Si la conversión falla, se debe lanzar una excepción NumberFormatException  
que será capturada y manejada adecuadamente  
*/
```

# FUNCTIONS

```
object WhatsAFunction extends App {  
  
    val doble = new MyFunction [Int, Int] {  
        ↑ apply  
        override def apply(elemento: Int): Int = elemento * 2  
    }  
  
    println (doble(2))  
  
}  
  
trait MyFunction[A, B] {  
    def apply(elemento: A):B  
}
```

Todas las funciones en Scala son objetos

# FUNCTIONS

```
//function types = Function1[A,B]

val cadenaAentero = new Function1[String, Int] {
    ↑ apply
    override def apply(string: String): Int = string.toInt
}

println(cadenaAentero("3") + 4)
```

# FUNCTIONS. Ejercicio 1

```
/*Ejercicio 1: Realiza una función con dos parámetros de tipo cadena  
y el resultado es su concatenación */
```

```
def concatenar: (String, String) => String = new Function2[String, String, String] {  
    ↑ apply  
    override def apply(a: String, b: String): String = a + b  
  
    println(concatenar ("Hola, ", "Scala"))
```

## FUNCTIONS. Ejercicio 2

```
/*Ejercicio 2: Define una función que toma como argumentos un entero y otra función que  
la cual toma otro entero y retorna otro entero  
| - cuál es el tipo de esta función  
| - cómo hacer esto  
*/
```

```
//Function1[Int, Function1[Int,Int]]  
val specialFunction: Function1[Int, Function1[Int, Int]] = new Function1[Int, Function1[Int, Int]] {  
    ↑ apply  
    | | override def apply(x: Int): Function1[Int, Int] = new Function1[Int, Int] {  
        ↑ apply  
        | | | | override def apply(y: Int): Int = x + y  
    }  
}
```

# FUNCTIONS. Declaración de Tipo Value

```
//Function1[Int, Function1[Int,Int]]  
val specialFunction: Function1[Int, Function1[Int,Int]] = new Function1[Int, Function1[Int,Int]] {  
    override def apply(x: Int): Function1[Int, Int] = new Function1[Int,Int] {  
        | | override def apply(y: Int): Int = x + y  
    }  
}
```

FUNCTION. Ejemplo: Function2 (A, B, R)  $\equiv$  (A, B)  $\Rightarrow$  R; Function3(A, B, C, R)  $\equiv$  (A, B, C)  $\Rightarrow$  R

```
val specialFunction: ((Int, Int) => Int) = new Function1[Int, Function1[Int,Int]] {  
    override def apply(x: Int): Function1[Int, Int] = new Function1[Int,Int] {  
        | | override def apply(y: Int): Int = x + y  
    }  
}
```

# FUNCIONES ANÓNIMAS

```
// función anónima  
val fDoble = (x: Int) => x * 2
```

```
val fDoble: Int => Int = x => x * 2
```

## MÚLTIPLES PARÁMETROS

```
//MÚLTIPLE PARÁMETROS
```

```
val suma = (a: Int, b: Int) => a + b
```

```
val suma: (Int, Int) => Int = (a: Int, b: Int) => a + b
```

Una **función anónima** es una función sin nombre que se define directamente como una expresión, también conocida como *lambda*.

*La función anónima se declara:*

- Los **parámetros** aparecen a la **izquierda** del =>
- El **cuerpo de la función** aparece a la **derecha** del =>

# FUNCIONES ANÓNIMAS

SIN PARÁMETROS. Misma definición

```
//SIN PARÁMETROS  
val haceAlgo = () => 3  
val haceAlgo: () => Int = () => 3
```

¿Imprime lo mismo por pantalla?

```
println(haceAlgo)  
println(haceAlgo())
```

# FUNCIONES ANÓNIMAS

La impresión que realiza es la siguiente:

```
println(haceAlgo)  
println(haceAlgo())
```



```
println(haceAlgo) //devuelve la instancia  
println(haceAlgo()) //devuelve el 3
```

Resultado de la ejecución:

```
WhatsAFunction$$Lambda$7/0x000000080109c7a0@71be98f5  
3
```

# FUNCIONES ANÓNIMAS

```
//más sintaxis  
val incrementa: Int => Int = _ + 1 // equivalente x = x + 1  
val sumar: (Int, Int) => Int = _ + _ //equivalente (a,b) = a + b
```

# HOF (HIGHER ORDER FUNCTION) . CURRIES

En Scala, currying es una técnica funcional que permite transformar una función que toma múltiples argumentos en una secuencia de funciones que toman un solo argumento cada una.

```
val superFuncion: (Int, (String, (Int => Boolean)) => Int)
```

1º parámetro de entrada un entero (**Int**, (String,...))

2º parámetro de entrada Función (Int, (**String**, (**Int => Boolean**)) => Int)

# HOF (HIGHER ORDER FUNCTION) . CURRIES

## ¿Qué imprime por consola?

Función que aplica una función n veces sobre el valor x

nVeces (f, n, x) --> f es la función, n número de veces, x es el valor

nVeces (f, 3, x) --> f(f(f(x))) = f(f, n-1, f(x))

```
def nVeces (f: Int => Int, n: Int, x: Int): Int =
  if (n <= 0) x
  else nVeces (f, n-1, f(x))

val unoMas = (x: Int) => x + 1
println(nVeces(unoMas, 10, 1))
```

# HOF (HIGHER ORDER FUNCTION) . CURRIES

Analiza la función y explica

```
//nvm(f,n) = x => f(f(f....(x)))
//increment10 = nvm(nVeces,10) = x => nVeces(nVeces....(x))
//val y = increment10(1)
def nVecesMejor(f: Int => Int, n: Int): (Int => Int) =
  if (n<=0) (x: Int) => x
  else (x: Int) => nVecesMejor(f, n-1) (f(x))

val mas10 = nVecesMejor(unoMas,10)
println(mas10(1))
```

# HOF (HIGHER ORDER FUNCTION) . CURRIES

```
//curried functions
val superAdder: Int => (Int => Int) = (x: Int) => (y: Int) => x + y
val add3 = superAdder(3) //y => 3 + y
println (add3(10))
```

## SuperAdder:

- Toma como entrada un entero Int y devuelve una función Int => Int y la función de salida toma como entrada un entero y devuelve como salida un entero.
- Función anónima (lambda):
  - (x: Int) indica que toma como parámetro un entero
  - (y: Int) => x + y indica que devuelve otra función que toma un entero como entrada y devuelve la suma de x + y

# HOF (HIGHER ORDER FUNCTION) . CURRIES

```
//curried functions
val superAdder: Int => (Int => Int) = (x: Int) => (y: Int) => x + y
val add3 = superAdder(3) //y => 3 + y
println (add3(10))
```

## add3:

- Función *superAdder* pasa el valor 3 como x y devuelve una nueva función (*y: Int*)=>  $3 + y$
- Esta nueva función, que llámamos *add3*, suma 3 a cualquier número que se le pase como argumento.

Por lo tanto, *add3* es una función que toma un entero y devuelve  $3 + y$

# HOF (HIGHER ORDER FUNCTION) . CURRIES

```
//curried functions
val superAdder: Int => (Int => Int) = (x: Int) => (y: Int) => x + y
val add3 = superAdder(3) //y => 3 + y
println (add3(10))
```

println(add3(10)):

- **superAdder** es una función de orden superior porque devuelve otra función.
- **add3** es el resultado de llamar a superAdder(3), lo que devuelve una función que suma 3 a su argumento.
- **add3(10)**, que calcular  $3 + 10 = 13$  y lo imprime

¿Imprime lo mismo que println(add3(10))?

```
println(superAdder(3)(10))
```

# HOF (HIGHER ORDER FUNCTION) . CURRIES. EJERCICIOS

```
//EJERCICIO 1
/* 1. Define una función llamada superMult que recibe:
   - Parámetro de entrada un número entero
   - Parámetro de salida devuelve una función que recibe un número entero y devuelve un número entero
 2. Define una función multPorCinco que use la función superMult para multiplicar por 5.
 3. Imprime por pantalla el resultado de llamar a multPorCinco pasándole como parámetro el valor 7.
*/

```

# HOF (HIGHER ORDER FUNCTION) . CURRIES. EJERCICIOS

```
//EJERCICIO 2
/* 1. Define una función llamada superConcatenar que recibe:
 - Parámetro de entrada una cadena.
 - Parámetro de salida devuelve otra función que devuelve otra cadena como sufijo.
 La función debe concatenar ambas cadenas con un espacio entre ellas.

2. Define una función que utilice la función superConcatener que siempre utilice el prefijo "Hola"

3. Imprime por pantalla a la función superConcatenar.

*/
```

# HOF (HIGHER ORDER FUNCTION) . CURRIES. EJERCICIOS

```
//EJERCICIO 3
/* Sistema de notificaciones que genera mensajes personalizados para los usuarios.
```

Implementa en Scala:

1. Definición de un saludo inicial como Hola o Estimado cliente.
2. Añadir el nombre del usuario
3. Personalizar el mensaje con un contenido específico.

Tareas a realizar:

1. Implementa una función `generaMensaje` con los siguientes parámetros de entrada:
  - tipo cadena para indicar el saludo.
  - tipo cadena para indicar el nombre
  - tipo cadena para indicar el mensaje

El resultado de la función debe devolver los tres parámetros en un solo mensaje con el siguiente formato:  
| saludo, nombre: mensaje

2. Usar la función `generaMensaje` para crear una nueva función `saludoCliente` que utilice el saludo "Estimado Cliente"
3. Genera un mensaje completo utilizando `saludoCliente` para un cliente llamado Carlos con el contenido "Su pedido ha sido enviado con éxito"
4. Genera e imprime otro mensaje directamente utilizando `generaMensaje` con los valores:  
| "Hola", "María", "Gracias por registrarse en nuestra plataforma"

```
*/
```

## MAP – FlatMAP – FILTER - FOR

```
val list = List(1,2,3)
println(list)
println(list.head)
println(list.tail)
```

- 1. val list = List (1,2,3): Lista inmutable en Scala con los elementos 1, 2 y 3.**
- 2. list.head: devuelve el primer elemento de la lista.**
- 3. list.tail: devuelve una nueva lista que contiene todos los elementos excepto el primero.**

# MAP – FlatMAP – FILTER - FOR

```
val list = List(1,2,3)  
//map  
println(list.map(_ +1))  
println(list.map(_+ " es un número"))
```

1. La función map aplica una operación a cada elemento de la lista y devuelve una nueva lista con los resultados.

2. ( $_ + 1$ ) es una función anónima que toma cada elemento de la lista y le suma 1. Resultado:

1 suma 1 = 2

2 suma 1 = 3



List(2, 3, 4)

3 suma 1 = 4

## MAP – FlatMAP – FILTER - FOR

```
val list = List(1,2,3)
//map
println(list.map(_ +1))
println(list.map(_+ " es un número"))
```

**3. list.map (\_ + “ es un número”): concatena el texto “es un número” a cada elemento de la lista.**

**1 se convierte “1 es un número”**

**2 se convierte “2 es un número”**

**3 se convierte “3 es un número”**

# MAP – FlatMAP – FILTER - FOR

```
val list = List(1,2,3)  
  
//filter  
println(list.filter(_ % 2 == 0))
```

**filter** se utiliza para filtrar elementos de una colección que cumplen una condición.

**\_ % 2 == 0:** función anónima (lambda) que verifica si un número es divisible entre 2.

**\_** representa cada elemento de la lista

**%** el operador residuo. En este caso dividir el número entre 2 es 0.

El resultado de filter es una nueva lista que contiene los números pares en este caso 2.

## MAP – FlatMAP – FILTER - FOR

```
val list = List(1,2,3)
//flatMap
val toPair = (x: Int) => List(x, x+1)
println(list.flatMap(toPair))
```

**list.flatMap(toPair):**

**flatMap:** aplica una función a cada elemento de una colección o contenedor y luego aplana los resultados en una sola colección o contenedor. Esto elimina la necesidad de manejar estructuras anidadas.

**toPair:** se aplica a cada elemento de la lista. Resultado: (1, 2, 2, 3, 3, 4)

## MAP – FlatMAP – FILTER - FOR

```
//imprime todas las combinaciones entre dos listas
val numero = List (1,2,3,4)
val caracteres = List ('a','b','c','d')
val colores = List ("claro", "oscuro")
//List ("a1","a2"...."d4")

//iteraciones
val combinaciones = numero.flatMap(n=> caracteres.map(c=> "" + c + n))
println(combinaciones)
```

# MAP – FlatMAP – FILTER - FOR

```
//imprime todas las combinaciones entre dos listas
val numero = List (1,2,3,4)
val caracteres = List ('a','b','c','d')
val colores = List ("claro", "oscuro")
//List ("a1","a2"...."d4")

//iteraciones
val combinaciones = numero.flatMap(n=> caracteres.map(c=> "" + c + n))
println(combinaciones)
```

Resultado:

```
List(a1, b1, c1, d1, a2, b2, c2, d2, a3, b3, c3, d3, a4, b4, c4, d4)
```

# MAP – FlatMAP – FILTER - FOR

```
//imprime todas las combinaciones entre dos listas
val numero = List (1,2,3,4)
val caracteres = List ('a','b','c','d')
val colores = List ("claro", "oscuro")
//List ("a1","a2"...."d4")
|
//iteraciones
val combinaciones = numero.map(n=> caracteres.map(c=> "" + c + n))
println(combinaciones)
```

**Resultado: ¿cómo lo devuelve?**

# MAP – FlatMAP – FILTER - FOR

```
//imprime todas las combinaciones entre dos listas
val numero = List (1,2,3,4)
val caracteres = List ('a','b','c','d')
val colores = List ("claro", "oscuro")
//List ("a1","a2"...."d4")
|
//iteraciones
val combinaciones = numero.map(n=> caracteres.map(c=> "" + c + n))
println(combinaciones)
```

Resultado:

```
List(List(a1, b1, c1, d1), List(a2, b2, c2, d2), List(a3, b3, c3, d3), List(a4, b4, c4, d4))
```

# **MAP - FlatMAP - FILTER - FOR**

## Resultado: ¿?

# MAP – FlatMAP – FILTER - FOR

```
//foreach  
list.foreach(println)
```

Resultado de Imprimir:

```
1  
2  
3
```

# MAP – FlatMAP – FILTER - FOR

```
//for-comprehensions
val numero = List (1,2,3,4)
val caracteres = List ('a','b','c','d')
val colores = List ("claro", "oscuro")

//for-comprehensions
val forCombinaciones = for {
    n <- numero
    c <- caracteres
    color <- colores

} yield "" + c + n + "-" + color
println (forCombinaciones)
```

# MAP – FlatMAP – FILTER - FOR

```
//for-comprehensions
val forCombinaciones = for {
    n <- numero
    c <- caracteres
    color <- colores

} yield "" + c + n + "-" + color
println (forCombinaciones)
```

**For{...} yield...:** se utiliza para generar una nueva colección aplicando combinaciones de elementos de las listas de entrada.

**N <- numero:** itera sobre cada elemento de la lista número.

**Yield:** especifica qué valor se generará para cada combinación.

# MAP – FlatMAP – FILTER - FOR

## Resultado de Imprimir:

```
List(a1-claro, a1-oscuro, b1-claro, b1-oscuro, c1-claro, c1-oscuro, d1-claro, d1-oscuro, a2-claro, a2-oscuro, b2-claro, b2-oscuro, c2-claro, c2-oscuro, d2-claro, d2-oscuro, a3-claro, a3-oscuro, b3-claro, b3-oscuro, c3-claro, c3-oscuro, d3-claro, d3-oscuro, a4-claro, a4-oscuro, b4-claro, b4-oscuro, c4-claro, c4-oscuro, d4-claro, d4-oscuro)
```

Resultado genera una cadena con el formato <c><n>-<color> y las acumula en una nueva lista.

# EJERCICIOS

/\* Ejercicio 1:

Dada una lista de alumnos y alumnas con sus nombres y una lista de asignaturas disponibles, genera todas las combinaciones posibles de ambas creando una cadena con el siguiente formato:  
"Nombre del estudiante" está inscrito en "Asignaturas"

\*/

/\*Ejercicio 2:

En una tienda de frutas, cada fruta tiene un precio por unidad. Se necesita realizar las siguientes operaciones sobre una lista de compras de un cliente:

1. Filtrar las frutas cuyo precio por unidad sea mayor a 3
2. Calcular el precio total para cada fruta filtrada considerando la cantidad comprada
3. Generar una lista de mensajes indicando cuánto se gastará por cada fruta en el formato:  
"Fruta: <nombre>, Total: <total>"

\*/

# SECUENCIAS (SEQUENCES)

```
//Seq  
val aSecuen = Seq(1,2,3,4)  
println(aSecuen) // devuelve List(1,2,3,4)  
println(aSecuen.reverse) // devuelve Lista(4,3,2,1)  
println(aSecuen(2)) //devuelve el elemento 3  
println(aSecuen ++ Seq(4,5,6)) // concatena ambas secuencias List (1,2,3,4,5,6)  
println(Seq(8,4,3).sorted) //devuelve la secuencia ordenada
```

Una interfaz muy general para estructuras de datos que

- tienen un orden bien definido
- pueden indexarse

Admite diversas operaciones:

- apply, iterator, length, reverse para indexar e iterar
- concatenation, appending, prepending
- muchas otras: grouping, sorting, zipping (comprimir), searching, slicing (trocear)

# SECUENCIAS (SEQUENCES). EJERCICIO

```
/*Ejercicio:  
Analiza los datos meteorológicos de la secuencia de temperaturas diarias (en grados Celsius) registradas durante una semana.  
Temperaturas de la semana: 815, 20, 18, 22, 16, 19, 21)  
Tareas a realizar:  
1.- Imprime por pantalla las temperaturas de la semana.  
2.- Encuentra la temperatura más alta de la semana.  
3.- Encuentra la temperatura más baja de la semana.  
4.- Calcula la temperatura promedio de la semana (usa .sum para sumar y divide por el número de días)  
5.- Ordena las temperaturas de menor a mayor e imprime la secuencia ordenada.  
6.- Agrega un registro de temperatura adicional (ejemplo 23 grados) al final de la secuencia y muestra la nueva lista.  
7.- Encuentra el número de días en que la temperatura fue mayor o igual a 20 grados.  
*/
```

# RANGOS (RANGES)

```
//Ranges
val aRange: Seq[Int] = 1 until 10
aRange.foreach(println) //devuelve 1  2  3 .... 9
(1 to 10).foreach(x=> println("Hola")) //imprime por pantalla Hola el rango definido
```

Un rango en Scala es una secuencia de números que sigue un patrón definido (por ejemplo, desde un número inicial hasta un número final).

Los rangos son útiles para iterar sobre valores consecutivos.

# RANGOS (RANGES). EJERCICIO

/\*Ejercicio:

Sistema debe generar horarios de clases para un día específico. Hay 10 clases en el día, y cada clase dura 1 hora, comenzando a las 8 de la mañana y terminando a las 17 de la tarde.

Resuelve las siguientes tareas utilizando Rangos:

- 1.- Genera un rango de horas para las clases, teniendo en cuenta que comienzan a las 8 (hora incluida) y terminan a las 17 (hora excluida) e imprime por pantalla.
- 2.- Asigna a cada una de las horas que hay clase el siguiente mensaje e imprime por pantalla "Clase en la hora X" --> cambia el valor X por la hora actual.
- 3.- Calcula el total de clases del día usando el rango y muestra el resultado. "Número total de clases: X" --> X el valor calculado
- 4.- Imprime por pantalla un mensaje cada dos horas, indicando que hay un descanso --< PISTA: (8 until 17 by 2) --> "by 2" avanza el rango de dos en dos

\*/

# LISTS

```
//Lists  
val aList = List (1,2,3)  
val prepended = 42 :: aList //añade a la lista como primer elemento el 42  
println(prepended)
```

```
//Lists  
val aList = List (1,2,3)  
val prepended = 42 +: aList :+ 89 //añade a la lista como primer elemento el 42 y como último elemento 89  
println(prepended)
```

```
val manzanas5 = List.fill(5) ("manzana")  
println(manzanas5) //imprime una lista de 5 elementos con el valor "manzana"
```

```
val aList2 = List(1,2,3)  
println(aList2.mkString("-| -")) //imprime los elementos de la lista separados -| === 1-| -2-| -3
```

**mkString** convierte los elementos de una colección en una sola cadena, separando los elementos con un delimitador definido entre paréntesis.

# ARRAY

```
//arrays
val numeros = Array (1,2,3,4)
val tresElementos = Array.ofDim[Int](3) // Array.ofDim[String](3)
tresElementos.foreach(println) //¿Qué imprime por pantalla? 0    0    0 // null    null    null

//mutation
numeros(2) = 0 //estamos actualizando el valor de la posición 2 a 0
println(numeros.mkString(" ")) //imprime por pantalla 1 2 0 4

💡//arrays y secuencias
val numerosSeq: Seq[Int] = numeros //conversión implícita
println(numerosSeq)
```

# ¿Cuándo utilizar “Array” o “List”?

**Mutabilidad.** Si necesitamos modificar los valores de los elementos con tamaño fijo entonces utilizamos **Array** porque **List** es inmutable (se usa más en programación funcional) no permitiendo cambiar los valores una vez creados .

```
//array
val arr = Array (1, 2, 3)
arr(0) = 10 //modificamos el valor del primer elemento
println(arr.mkString(", "))
```

**Estructura.** Si necesitamos una colección fija de elementos de tamaño predefinido con acceso rápido a los elementos entonces utilizamos **Array**. En **List** cada elemento apunta al siguiente y el último apunta a una lista vacía y acceder a un elemento específico por índice es más lento.

# EJERCICIOS: ¿Array o List?

```
/* Ejercicios: Resuelve los siguientes ejercicios utilizando ¿Array o List? y el por qué.

1.- Implementa una aplicación que almacene las lecturas de un sensor de temperatura. Cada vez que llega una nueva lectura, se debe agregar en la colección sin eliminar las lecturas anteriores.
2.- Necesitamos almacenar las puntuaciones de 6 jugadores en una partida, donde las puntuaciones deben ser accesibles para calcular su promedio al final.
3.- En una lista de compras, los productos comprados deben almacenarse en el orden en el que se van comprando.
4.- Tenemos un inventario donde los elementos pueden cambiar de cantidad rápidamente.
*/
```

# VECTOR

- **Inmutabilidad:** no puedes cambiar sus elementos directamente. Para modificar los valores de un Vector como *update* devuelve un nuevo vector con los cambios aplicados, sin modificar el vector original.
- **Estructura basada en 32 ramas**, dividiendo los datos en bloques pequeños permitiendo el acceso por índice ( $O(\log_{32}(n))$ ) y actualizaciones eficientes. Representación podría ser:

Nivel 1: [ Nodo\_1, Nodo\_2, Nodo\_3, ... ] // Raíz del árbol (32 referencias)  
Nivel 2: [ Elementos 1-32, Elementos 33-64, ..., Elementos 993-1000 ]

- **Acceso rápido** pero no tan rápido como Array.
- **Es dinámico** porque te permite agregar o modificar obteniendo una nueva colección con los cambios.

```
//Vector (Inmutable)
val vector = Vector(1, 2, 3)
```

# Comparar Vector vs Array

```
//Vector (Inmutable)
val vector = Vector(1, 2, 3)
val updatedVector = vector.updated(1, 42) // Modificar un elemento devuelve un nuevo Vector
println(vector)          // Vector(1, 2, 3) - Sin cambios
println(updatedVector) // Vector(1, 42, 3) - Nuevo Vector

//Array (mutable)
val array = Array(1, 2, 3)
array(1) = 42 //Modificar un elemento cambia el Array original
| println(array.mkString(", ")) // 1, 42, 3
```

## RESUMEN:

- **Array**: Ideal para trabajar con datos mutables y de tamaño fijo.
- **List**: Ideal para operaciones funcionales y cuando las inserciones al inicio son frecuentes.
- **Vector**: Opción predeterminada para colecciones inmutables con buen rendimiento general.

# Ejercicio

En Aules, disponéis de un ejemplo de código en Scala donde el objetivo del ejercicio es analizar y realizar una breve explicación del resultado al ejecutar el código de Scala.

“15\_Ejercicio\_Vector.scala”

# TUPLAS. LISTAS FINITAS Y ORDENADAS

```
//lists. Listas finitas y ordenadas
val aTuple = new Tuple2 (2, "Hola, Scala") // Tuple2 [Int, String] = (Int, String)
| val aTuple2 = (2, "Hola, Scala2") //sin la palabra new

println(aTuple._1) //2
println(aTuple.copy(_2 ="Adiós Java"))
println(aTuple.swap) //intercambia los elementos
```

- **Declaración:** con la palabra reservada “new” sin new. Tenemos hasta 22 elementos de una tupla.
- **nombre\_tupla.\_x:** donde nombre\_tupla es el nombre declarado y x es la posición del elemento que queremos obtener.
- **copy:** función que nos permite copiar todos los elementos de una tupla o asignar otro valor a un elemento \_x =”...” donde x es la posición del elemento que queremos cambiar.
- **swap:** intercambia la posición de los elementos. Es decir, tupla (2, “Hola,Scala”) pasa a ser (“Hola, Scala”,2)

# MAPS

```
//Maps: son colecciones que se usan para asociar unas con otras cosas  
//Maps : keys --> values  
val aMap: Map[String, Int] = Map()  
val dirTelefono = Map(("Jim", 999), "Daniel" -> 777)  
//a-> b === (a, b)  
println(dirTelefono)
```

```
//operaciones básica Maps  
println(dirTelefono.contains("Jim")) //devuelve verdadero o falso si contiene la clave  
println(dirTelefono("Jim")) //devuelve el valor asociado a la clave  
println(dirTelefono("maria")) //¿qué sucede?
```

# MAPS

```
val dirTelefono = Map(("Jim", 999), "Daniel" -> 777)  
println(dirTelefono("maria")) //¿qué sucede?
```

Si intentamos imprimir un elemento de Map que no existe el resultado que devuelve es un error

```
Exception in thread "main" java.lang.ExceptionInInitializerError  
    at TuplesAndMaps.main(16_TuplesAndMaps.scala)  
Caused by: java.util.NoSuchElementException: key not found: maria  
    at scala.collection.immutable.Map$Map2.apply(Map.scala:316)  
    at TuplesAndMaps$.<clinit>(16_TuplesAndMaps.scala:21)  
    ... 1 more
```

Para evitar que muestre error añadimos a continuación .withDefaultValue(-1)

```
val dirTelefono = Map(("Jim", 999), "Daniel" -> 777).withDefaultValue(-1) //para evitar errores  
println(dirTelefono("maria")) //¿qué sucede?
```

# MAPS

Para añadir un nuevo emparejamiento a Map:

```
//añadir un emparejamiento
val nuevoEmpar = "Maria" -> 789
val dirTelefono2 = dirTelefono + nuevoEmpar
println(dirTelefono2)
```

# MAPS

## Funciones Map:

```
//funciones en mapas
//map, flatMap, filter
println(dirTelefono.map(pair => pair._1.toLowerCase -> pair._2)) //jim y daniel se imprimen en minúsculas
//filterkeys
println(dirTelefono.filterKeys(x => x.startsWith("J")))) //sintaxis lambda

//maps Values
println(dirTelefono.mapValues(number => number * 10)) //pasará por todos los valores y les agregar un 0

println(dirTelefono.mapValues (number => "789-" + number)) //¿qué pasará?
```

# MAPS

## Conversiones a otras colecciones:

```
//conversiones a otras colecciones
val dirTelefono = Map(("Jim", 999), "Daniel" -> 777).withDefaultValue(-1) //para evitar errores
println(dirTelefono.toList) //pasar a lista
```

```
val nombres = List("Carlos", "Miguel", "Manuel", "Roberto", "David", "Guillermo", "Daniel")
println(nombres.groupBy(name => name.charAt(0))) //agrupa los nombres por lista donde la letra inicial sea la misma
```

# EJERCICIOS

/\*Ejercicio 1

En este ejercicio 1 programa una lista de números enteros y devuelva una nueva lista en la que cada número sea el cuadrado del número original. Utiliza la función map para transformar la lista.

\*/

/\* Ejercicio 2

En este ejercicio 2 programa una lista de palabras y devuelva una nueva lista en la que cada palabra esté convertida a mayúsculas. Utiliza la función map para transformar la lista.

\*/

/\* Ejercicio 3

En este ejercicio 3 programa una lista de nombres completos (formados por nombre y apellido, separados por un espacio) y devuelva una nueva lista donde cada nombre esté en formato "Apellido, Nombre".

Utiliza la función map para realizar esta transformación.

\*/

# EJERCICIOS

/\*Ejercicio 1

En este ejercicio 1 programa una lista de números enteros y devuelva una nueva lista en la que cada número sea el cuadrado del número original. Utiliza la función map para transformar la lista.

\*/

/\* Ejercicio 2

En este ejercicio 2 programa una lista de palabras y devuelva una nueva lista en la que cada palabra esté convertida a mayúsculas. Utiliza la función map para transformar la lista.

\*/

/\* Ejercicio 3

En este ejercicio 3 programa una lista de nombres completos (formados por nombre y apellido, separados por un espacio) y devuelva una nueva lista donde cada nombre esté en formato "Apellido, Nombre". Utiliza la función map para realizar esta transformación.

\*/

# OPTION

**Option** permite manejar valores que pueden o no estar presentes, eliminando la necesidad de manejar valores nulos explícitamente. **Option** :

- **Some(valor)**: Representa un valor existente.
- **None**: Representa la ausencia de valor.

## Ventajas:

- Evita errores de referencia nula (NullPointerException).
- Hace que el manejo de valores ausentes sea explícito.
- Facilita el uso de operaciones funcionales como map, flatMap, y filter.

## Analogía:

Piensa en Option como una caja:

- Si hay algo dentro de la caja, está envuelto en Some(valor).
- Si la caja está vacía, representa None

# OPTION

En Aules, está el fichero de ***17\_Ejemplo\_Scala\_Option*** para revisar.

# EJERCICIO

/\* Ejercicio

Tenemos un sistema que almacena información de usuarios en una base de datos. La información se representa de la siguiente manera:

- Map [String, Map [String, String]] donde:
  - El primer parámetro String corresponde al Identificador del usuario : "IdUsuario"
  - El segundo parámetro Map [String, String] corresponde a los siguientes datos:
    - Nombre del usuario "nombre"
    - Saldo de la cuenta bancaria "saldo"

Realizar un método que tenga como parámetro de entrada el Identificador del usuario y que realice las siguientes acciones:

- 1.- Buscar el IdUsuario del sistema
- 2.- Si el IdUsuario existe, debéis extraer el nombre y el saldo del usuario
- 3.- Si tiene saldo, entonces debes convertir ese saldo a un valor numérico para aumentarlo 10%.
- 4.- Imprime por pantalla el nombre del usuario con el nuevo saldo obtenido. En caso de que el usuario no exista, o no haya datos no debes imprimir ningún resultado.

Datos del ejercicio:

```
(IdUsuario, nombre, saldo) --> ("1", "Carlos", "100")
(IdUsuario, nombre, saldo) --> ("2", "Ana", "200")
(IdUsuario, nombre, saldo) --> ("3", "Luis") --> OJO: NO TIENE SALDO
*/

```

# EJERCICIOS

```
/* Ejercicio 1:  
   Dado un mapa de productos, escribe una función que reciba el Identificador de un producto y devuelva la descripción.  
   Si no se encuentra, devuelve la siguiente descripción "Producto no disponible"  
*/  
  
/*Ejercicio 2  
   Crea una función que reciba un valor de tipo Any y determine si es un número, una cadena, un boolean o algo más.  
   Imprime por pantalla una descripción que indique el tipo.  
*/  
  
/*Ejercicio 3  
   Crea una función que reciba una tupla anidada del tipo (Int, (Int, Int)) y devuelva la suma de todos los números  
*/
```

# EJERCICIOS

/\*Ejercicio 3:

En una empresa que organiza eventos, se cuenta con una base de datos en formato de lista de mapas donde cada entrada representa a un asistente e incluye información como nombre, edad y correo electrónico.

Se requiere procesar esta información para obtener una lista de asistentes

mayores de edad con sus nombres en mayúsculas y correos electrónicos asociados.

La solución debe manejar posibles datos faltantes y permitir pruebas para verificar diferentes escenarios.

# EJERCICIOS

```
/* Ejercicio 4:  
 Una tienda en línea maneja un catálogo de productos almacenados en una lista de mapas, donde cada producto tiene atributos como nombre,  
 precio y categoría. Se requiere procesar esta información para obtener una lista de productos  
 cuyo precio sea mayor a 50, formateando sus nombres en mayúsculas y asociándolos con su categoría.  
 Es necesario manejar la posible ausencia de valores y garantizar que el procesamiento se realice correctamente mediante pruebas.  
 */
```