

# HCL (BIG DATA)

TAREA I UDI



---

ALUMNO CESUR

25/26

Alejandro Muñoz de la Sierra

PROFESOR

Óscar González Núñez

# INTRODUCCION

Adentrándonos en Scala, este trabajo explora cómo este lenguaje fusiona la programación funcional y la orientada a objetos, una combinación que, en mi opinión, ofrece una versatilidad considerable para el desarrollo de aplicaciones contemporáneas. Nos hemos propuesto aplicar conceptos clave de Scala, tales como objetos singleton, funciones inmutables, listas, recursión y el útil pattern matching, a un problema de programación bien conocido: el algoritmo de ordenación por burbuja.

Este ejercicio no solo nos expuso a la sintaxis del lenguaje; de hecho, nos brindó la oportunidad de poner en práctica buenas prácticas de programación funcional. Aprendimos a manejar datos inmutables y a construir nuevas estructuras de datos sin alterar las originales, algo fundamental. Además, nos demostró cómo traducir algoritmos tradicionales, a menudo concebidos de manera procedural, a un estilo funcional, más declarativo y, en mi experiencia, más expresivo, donde cada paso se vuelve más transparente y el flujo de datos se mantiene ordenado.

A lo largo del desarrollo, se consolidó la idea de que la programación funcional representa más que simplemente una forma diferente de codificar; es una perspectiva distinta sobre los datos y las operaciones que se llevan a cabo con ellos. Descubrimos cómo la recursión puede reemplazar a los bucles convencionales, cómo el pattern matching simplifica la desestructuración de listas, y cómo los objetos singleton permiten una organización centralizada y accesible de funciones y valores.

En las siguientes secciones, detallaremos paso a paso la implementación del algoritmo, explicando el código y compartiendo nuestras reflexiones sobre cada concepto que fuimos aprendiendo. Acompañaremos todo esto con ejemplos prácticos y enlaces a recursos oficiales y educativos. Esperamos que esta práctica sirva como una guía completa y clara para abordar problemas clásicos de programación desde un enfoque funcional en Scala.

# COMO FUNCIONA SCALA

## 1. Programación Funcional:

La programación funcional se fundamenta en un principio que, a mi modo de ver, resulta bastante elegante: la construcción de programas a partir de funciones puras. Cada función recibe datos de entrada, realiza cálculos y devuelve un resultado, sin alterar nada fuera de su ámbito. En esencia, evita las variables globales y los efectos secundarios.

En este paradigma, se evita el uso de variables mutables, aquellas cuyo valor cambia con el tiempo. En cambio, se prioriza el uso de estructuras inmutables y el flujo de datos a través de funciones.

Este enfoque ofrece varias ventajas:

El código se vuelve más predecible y, por ende, más fácil de probar, ya que una función siempre devuelve el mismo resultado para los mismos datos de entrada.

Es más seguro en entornos concurrentes, donde varios procesos o hilos pueden ejecutarse simultáneamente.

Además, el código tiende a ser más expresivo, ya que muchas operaciones se pueden expresar de forma directa mediante funciones como map, filter o reduce.

En resumen, la programación funcional busca crear programas más limpios, confiables y, en última instancia, más fáciles de mantener.

## 2. La Recursión:

En la programación funcional, los bucles tradicionales como `for` o `while` no son bien recibidos. La razón es que los bucles implican modificar una variable contadora, lo que se considera un efecto secundario, algo que este paradigma busca evitar.

En cambio, se utiliza la recursión, que consiste en que una función se invoca a sí misma hasta que se cumple una condición de parada. De esta forma, la repetición se logra sin modificar variables, manteniendo la pureza de las funciones.

Podríamos decir que la recursión cumple la misma función que los bucles en los lenguajes imperativos, pero con un enfoque más declarativo: en lugar de indicar cómo repetir una acción, simplemente definimos cuándo debe detenerse la repetición.

Este estilo, además de ser más coherente con la filosofía funcional, permite escribir código más elegante y, en muchos casos, más fácil de razonar, aunque a veces pueda parecer un poco "enrevesado" al principio.

# 01

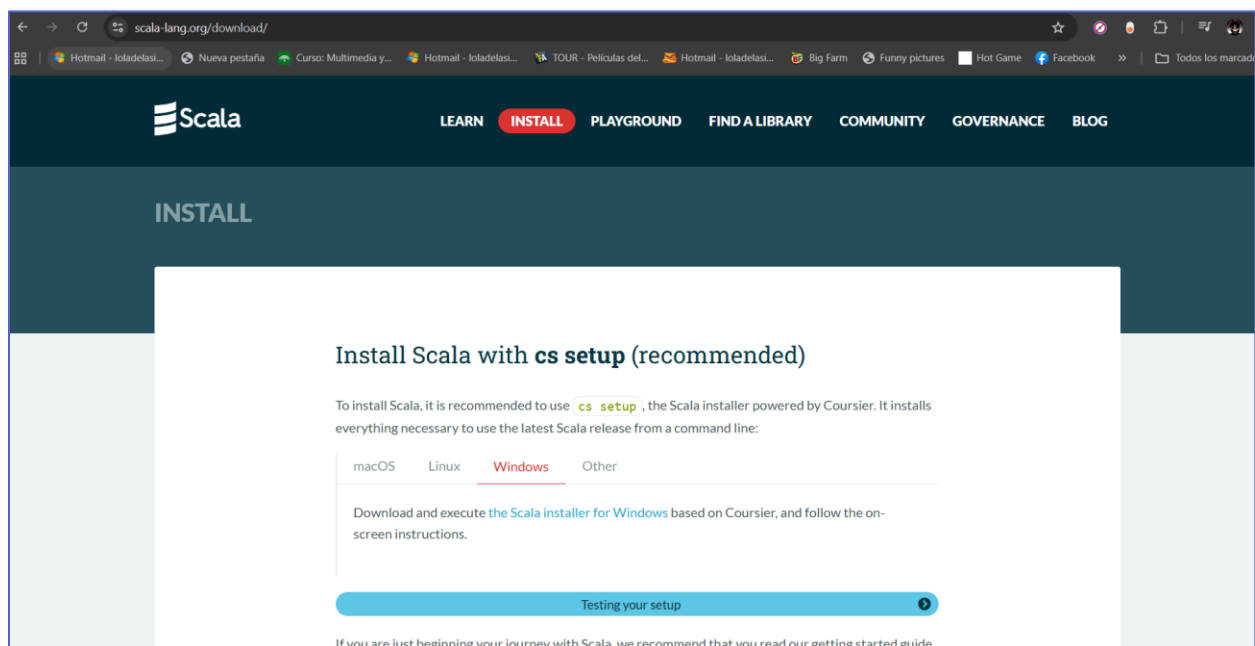
## PREPARAR EL ENTORNO (JDK Y SCALA)

Si ya tenemos Java instalado, debido a que estuvimos usando JDK en el curso anterior, por lo tanto no es necesario volverlo a descargar. Sólo necesitaremos el compilador e intérprete de Scala.

Esta compatibilidad hace que Scala sea una opción muy atractiva para quienes ya saben Java, porque les permite usar sus conocimientos y las mismas herramientas.

Para empezar con Scala, el primer paso es visitar la página oficial:

<https://www.scala-lang.org/download/>



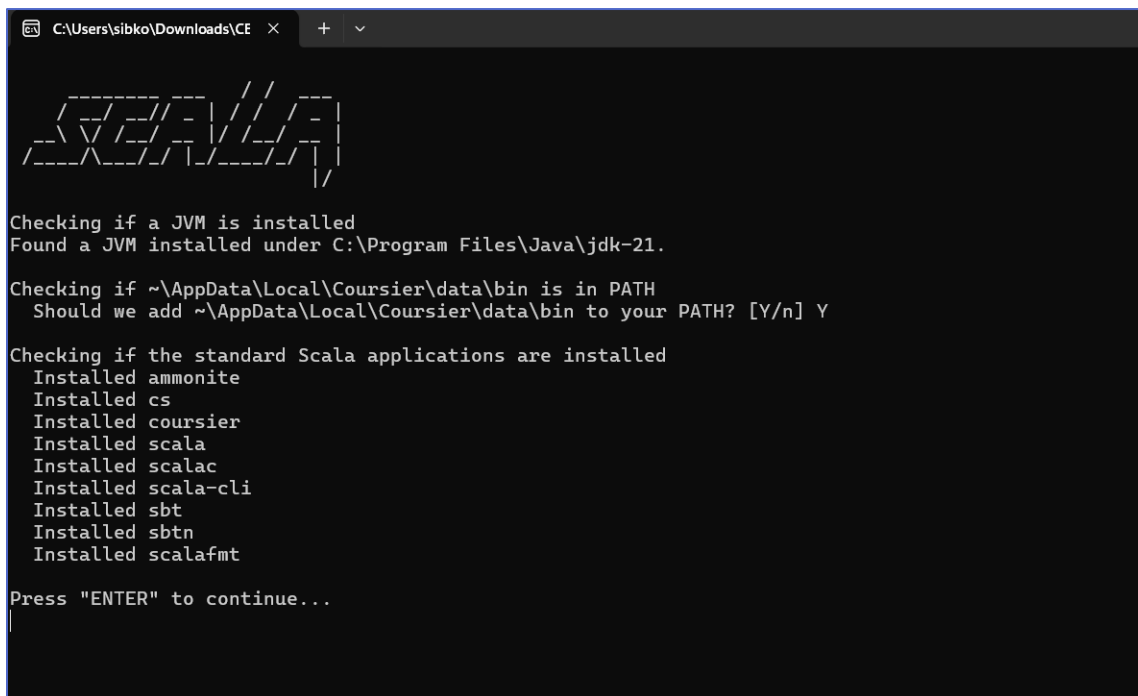
Allí encontraremos la versión Scala 3.x, que suele ser la más estable.

Al instalar, podemos elegir entre dos opciones:

Scala CLI, una opción más moderna que incluye todo lo necesario para empezar (compilador, intérprete, etc.).

La distribución binaria, que es un enfoque más tradicional.

En Windows, yo recomiendo usar Scala CLI, porque facilita mucho la instalación al incluir todo en un solo paquete.



```
C:\Users\sibko\Downloads\CE x + v

Checking if a JVM is installed
Found a JVM installed under C:\Program Files\Java\jdk-21.

Checking if ~\AppData\Local\Coursier\data\bin is in PATH
Should we add ~\AppData\Local\Coursier\data\bin to your PATH? [Y/n] Y

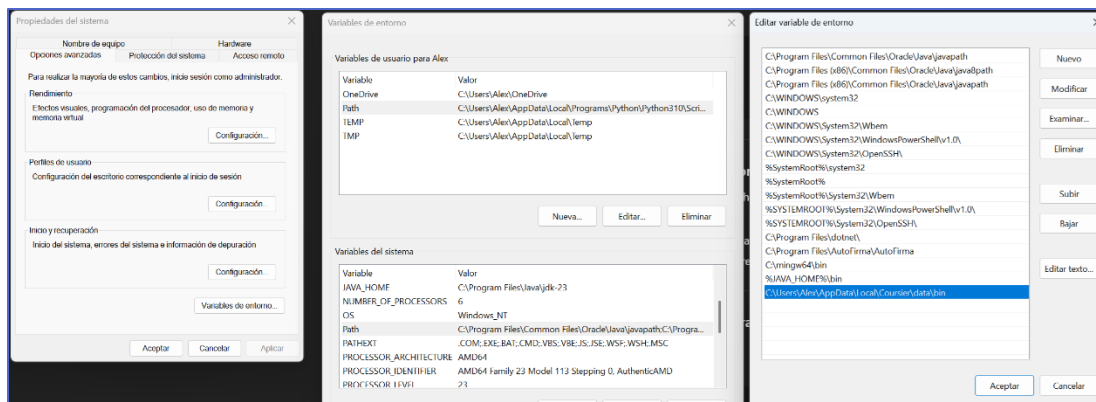
Checking if the standard Scala applications are installed
Installed ammonite
Installed cs
Installed coursier
Installed scala
Installed scalac
Installed scala-cli
Installed sbt
Installed sbtln
Installed scalafmt

Press "ENTER" to continue...
```

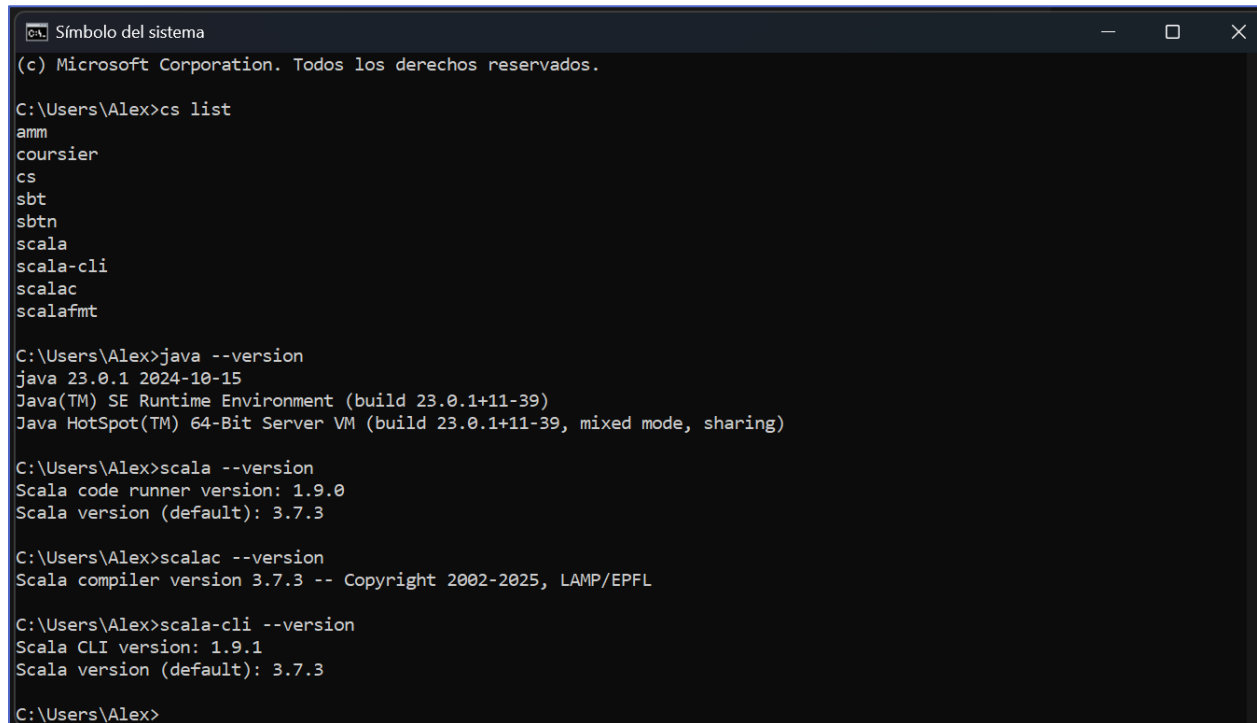
Después de la instalación, es buena idea verificar que todo funciona bien abriendo una terminal y comprobando que el sistema reconoce los comandos.

A veces, es necesario añadir la carpeta bin de Scala al PATH del sistema, como hicimos con Java.

Esto asegura que podamos ejecutar los comandos de Scala desde cualquier lugar de la terminal.



Para confirmar que la instalación se reconoció bien, usamos el comando:



```
Símbolo del sistema
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Alex>cs list
amm
coursier
cs
sbt
sbtn
scala
scala-cli
scalac
scalafmt

C:\Users\Alex>java --version
java 23.0.1 2024-10-15
Java(TM) SE Runtime Environment (build 23.0.1+11-39)
Java HotSpot(TM) 64-Bit Server VM (build 23.0.1+11-39, mixed mode, sharing)

C:\Users\Alex>scala --version
Scala code runner version: 1.9.0
Scala version (default): 3.7.3

C:\Users\Alex>scalac --version
Scala compiler version 3.7.3 -- Copyright 2002-2025, LAMP/EPFL

C:\Users\Alex>scala-cli --version
Scala CLI version: 1.9.1
Scala version (default): 3.7.3

C:\Users\Alex>
```

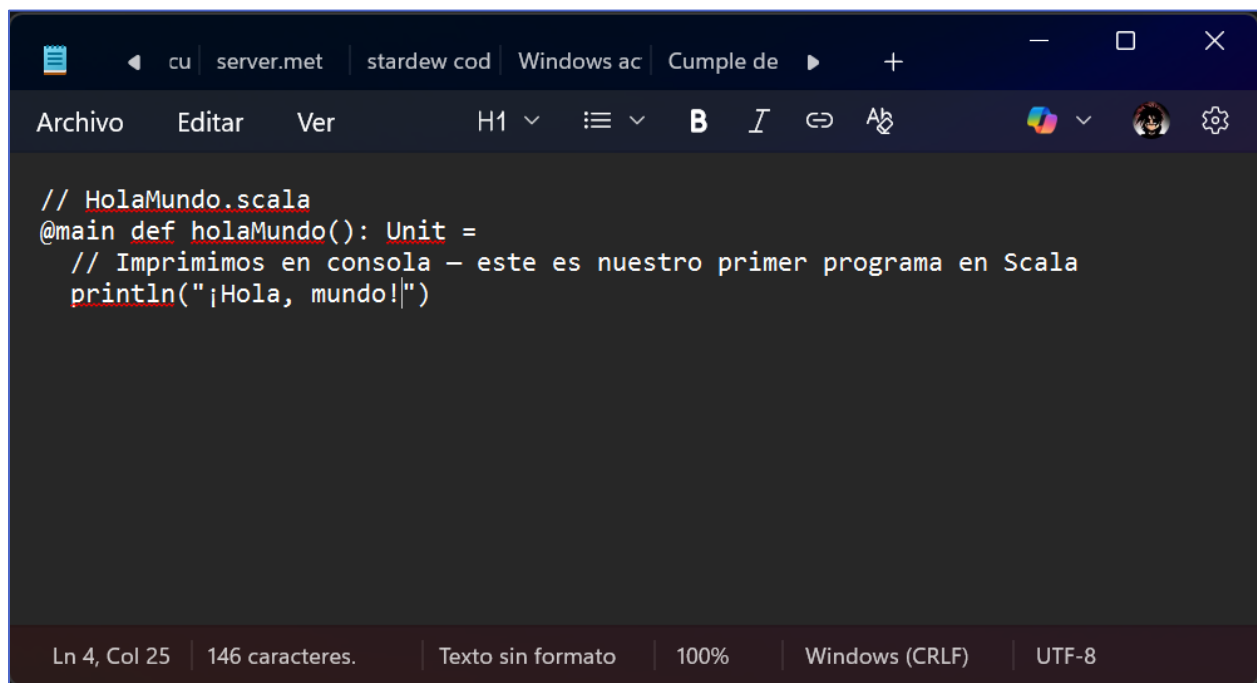
Si todo está correcto, el sistema mostrará la versión de Scala instalada, confirmando que podemos empezar a programar.

## 0 2

# “ H O L A M U N D O ” E N S C A L A

En este caso como es la primera vez que tocamos este lenguaje, mantenemos la introducción al mismo desde la base.

Creamos el código en un archivo de texto simple y le añadimos las extensión .scala, el interprete y el compilador de Scala reconocerán el archivo y si el código es correcto, el archivo se compilará y podrá ejecutarse.

A screenshot of a code editor window with a dark theme. The window has a title bar with several tabs: 'cu', 'server.met', 'stardew cod', 'Windows ac', and 'Cumple de'. Below the title bar is a menu bar with 'Archivo', 'Editar', and 'Ver'. To the right of the menu bar are icons for font size (H1), line numbers, bold (B), italic (I), undo (↶), redo (↷), and a search icon. The main area of the editor contains the following Scala code:

```
// HolaMundo.scala
@main def holaMundo(): Unit =
  // Imprimimos en consola – este es nuestro primer programa en Scala
  println("¡Hola, mundo!")
```

At the bottom of the editor, there is a status bar with the following information: 'Ln 4, Col 25', '146 caracteres.', 'Texto sin formato', '100%', 'Windows (CRLF)', and 'UTF-8'.

En este caso al ser un lenguaje nuevo la sintaxis cambia un poco. Algunos conceptos de Scala sería:

**@main** le estamos diciendo al programa que estamos ejecutando main, por lo que todo el código que escribamos después se ejecutará.

**def holamundo():** es la forma de definir funciones en scala, utilizando dos puntos como delimitador en lugar de llaves.



**Unit** = es el tipo que devuelve la función , lo que equivale al tipo de return. En este caso el tipo Unit equivale a una función tipo void, que sólo ejecuta código sin devolver nada.

**Println** , funciona igual que en Java.

Ahora sólo tenemos que compilar y ejecutar a la vez el archivo con scala-cli y obtener el resultado:

```
Símbolo del sistema - scala-cli run HolaMundo.scala
Microsoft Windows [Versión 10.0.26100.6725]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Alex>cd C:\Users\Alex\Documents\GitHub\Cesur-DAM\Asignaturas 2\HCL (Big Data)\UD1\Tarea 1

C:\Users\Alex\Documents\GitHub\Cesur-DAM\Asignaturas 2\HCL (Big Data)\UD1\Tarea 1>dir
El volumen de la unidad C no tiene etiqueta.
El número de serie del volumen es: D8CE-5356

Directorio de C:\Users\Alex\Documents\GitHub\Cesur-DAM\Asignaturas 2\HCL (Big Data)\UD1\Tarea 1

14/10/2025  08:04    <DIR>          .
14/10/2025  07:19    <DIR>          ..
14/10/2025  07:19             128.564 Captura de pantalla 2025-10-13 193350.png
14/10/2025  07:19             28.630 Captura de pantalla 2025-10-13 193556.png
14/10/2025  07:19             52.251 Captura de pantalla 2025-10-13 193800.png
14/10/2025  07:19             75.900 Caso1_UD1.docx
14/10/2025  08:04             153 HolaMundo.scala
14/10/2025  08:01              0 HolaMundo.txt
14/10/2025  07:19          257.992 Muñoz_de_la_Sierra_Alejandro_Plantilla_Big_Data_UD1_Tarea1 (Recuperado automáticamente).docx
14/10/2025  07:19          258.046 Muñoz_de_la_Sierra_Alejandro_Plantilla_Big_Data_UD1_Tarea1.docx
14/10/2025  07:42          341.089 path.png
                9 archivos          1.142.625 bytes
                2 dirs 160.099.209.216 bytes libres

C:\Users\Alex\Documents\GitHub\Cesur-DAM\Asignaturas 2\HCL (Big Data)\UD1\Tarea 1>scala-cli run HolaMundo.scala
Downloading compilation server 2.0.13
https://repo.scala-lang.org/artifactory/maven-nightlies/org/eclipse/ee4j/project/1.0.7/project-1.0.7.pom
100.0% [#####] 13.8 KiB (62.7 KiB / s)
https://repo.scala-lang.org/artifactory/maven-nightlies/org/eclipse/ee4j/project/1.0.7/project-1.0.7.pom.sha1
0.0% [ ] 0B (0B / s)
```

```
C:\Users\Alex\Documents\GitHub\Cesur-DAM\Asignaturas 2\HCL (Big Data)\UD1\Tarea 1>scala-cli run HolaMundo.scala
Downloading compilation server 2.0.13
Starting compilation server
Downloading Scala 3.7.3 bridge
Compiling project (Scala 3.7.3, JVM (23))
Compiled project (Scala 3.7.3, JVM (23))
¡Hola, mundo!

C:\Users\Alex\Documents\GitHub\Cesur-DAM\Asignaturas 2\HCL (Big Data)\UD1\Tarea 1>
```

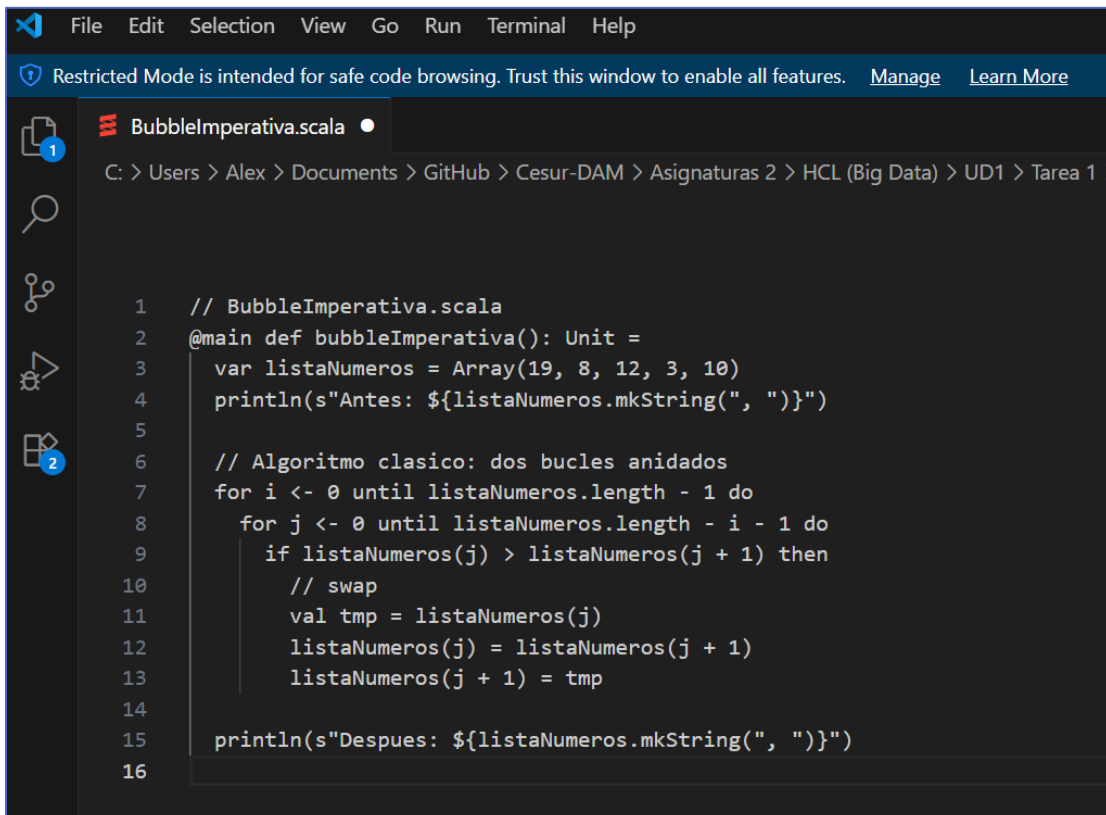
Vemos que la Consola imprime ¡Hola, mundo!

# ALGORITMO: ORDENACIÓN POR BURBUJA (BUBBLE SORT)

La ordenación por burbuja, es un algoritmo fundamental para ordenar listas. Básicamente, damos varias vueltas a la lista, comparando elementos que están uno al lado del otro y los intercambiamos si no están en el orden correcto. La elegimos, sobre todo, por su sencillez. Es fácil pillar la idea y nos permite ver cómo una misma idea se puede expresar de dos maneras muy diferentes: la forma imperativa, más tradicional, y la forma funcional más propia de Scala.

Para este algoritmo no usamos el editor de texto, si no algún IDE que tenemos instalado para ayudarnos a localizar errores y facilitar la edición. En nuestro caso usamos **VsCode**.

## 3A) Versión imperativa: Entendiendo el algoritmo



```

1 // BubbleImperativa.scala
2 @main def bubbleImperativa(): Unit =
3   var listaNumeros = Array(19, 8, 12, 3, 10)
4   println(s"Antes: ${listaNumeros.mkString(", ")}")
5
6   // Algoritmo clasico: dos bucles anidados
7   for i <- 0 until listaNumeros.length - 1 do
8     for j <- 0 until listaNumeros.length - i - 1 do
9       if listaNumeros(j) > listaNumeros(j + 1) then
10         // swap
11         val tmp = listaNumeros(j)
12         listaNumeros(j) = listaNumeros(j + 1)
13         listaNumeros(j + 1) = tmp
14
15   println(s"Despues: ${listaNumeros.mkString(", ")}")
16
  
```

Primero, creamos un Array mutable con los valores que queremos ordenar.

Luego, hacemos n-1 pasadas por la lista. En cada pasada, el valor más grande, como si fuera una burbuja, "sube" hacia el final.

En cada comparación, usamos un ``if`` y, si el par está desordenado, aplicamos un ``swap`` de valores.

Usamos un `println(s"..")` que nos permite añadir variables dentro del String de las comillas sin tener que estar poniendo `+`.

Usamos también el método `mkString(",")` que lo que hace es convertir una lista en un String, para imprimir todos los números sin tener que hacer un bucle.

Repetimos todo hasta completar las pasadas o hasta que en una pasada no haya cambios.

Con este enfoque, vemos clarísimamente cómo funciona todo: índices, bucles anidados y cómo va cambiando el arreglo. Es la forma que mejor muestra el comportamiento interno del algoritmo, y por eso la vemos primero. Nos ayuda a entender qué pasa en cada iteración antes de pasar a una versión sin efectos secundarios.

Compilamos, ejecutamos y vemos si el resultado es correcto:

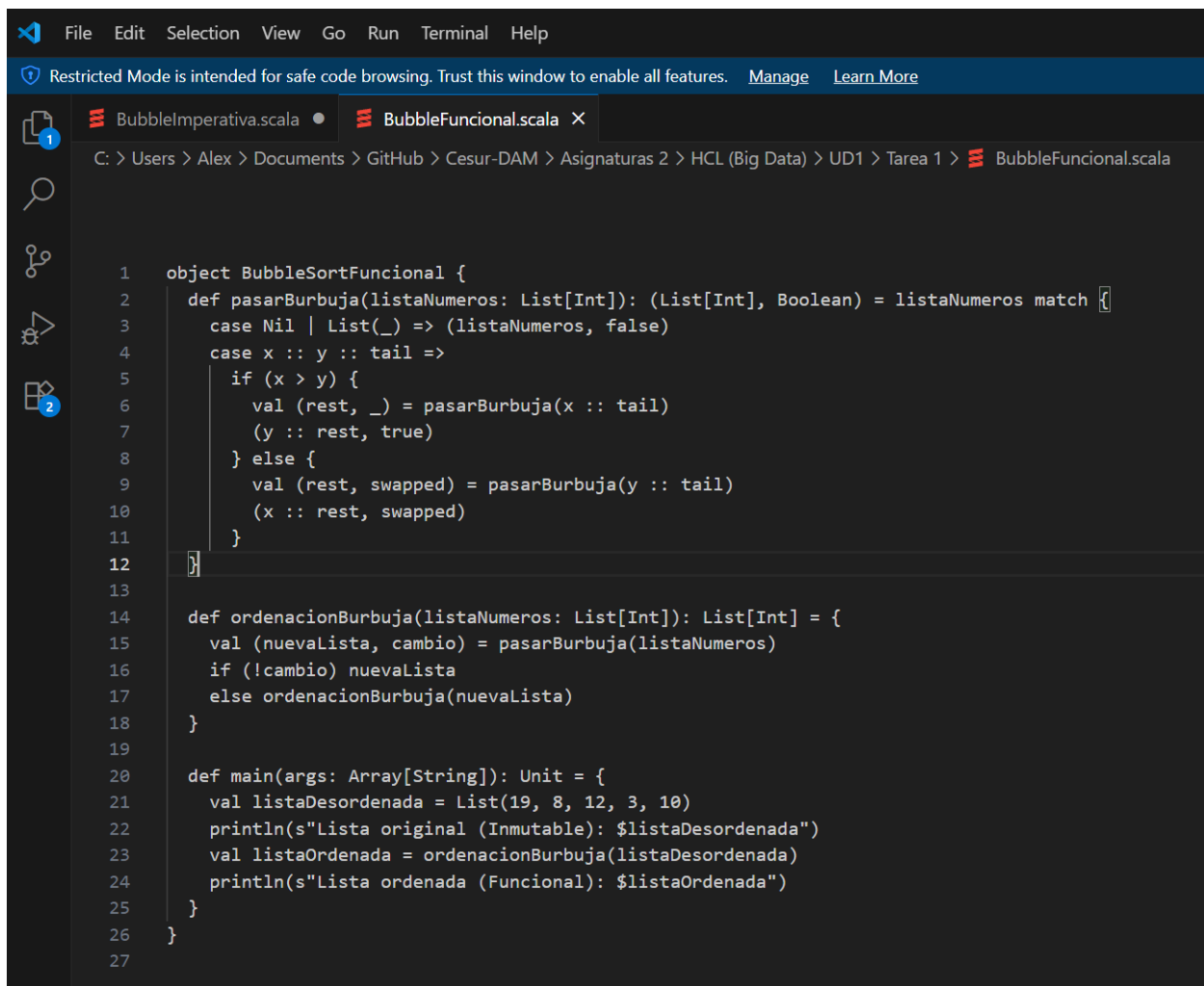
```
C:\Users\Alex\Documents\GitHub\Cesur-DAM\Asignaturas 2\HCL (Big Data)\UD1\Tarea 1>scala-cli run BubbleImperativa.scala
Compiling project (Scala 3.7.3, JVM (23))
Compiled project (Scala 3.7.3, JVM (23))
[E] Exiting BSP server with Connection reset
Bloop 'bsp' command exited with code 1. Something may be wrong with the current configuration.
Running the clean sub-command to clear the working directory and remove caches might help.
If the error persists, please report the issue as a bug and attach a log with increased verbosity by passing -v -v -v.
Antes: 19, 8, 12, 3, 10
Después: 3, 8, 10, 12, 19

C:\Users\Alex\Documents\GitHub\Cesur-DAM\Asignaturas 2\HCL (Big Data)\UD1\Tarea 1>
```

### 3B) Versión funcional: Inmutable y recursiva

¿Cómo lo planteamos?

Buscamos una versión que no cambie la colección original, sino que nos dé una nueva colección, pero ordenada. Para eso, usamos estructuras inmutables (como las `List`) y una estrategia basada en pasadas que crean listas nuevas. Repetimos estas pasadas hasta que no haya ningún intercambio.



```
1 object BubbleSortFuncional {
2   def pasarBurbuja(listaNumeros: List[Int]): (List[Int], Boolean) = listaNumeros match {
3     case Nil | List(_) => (listaNumeros, false)
4     case x :: y :: tail =>
5       if (x > y) {
6         val (rest, _) = pasarBurbuja(x :: tail)
7         (y :: rest, true)
8       } else {
9         val (rest, swapped) = pasarBurbuja(y :: tail)
10        (x :: rest, swapped)
11      }
12   }
13
14   def ordenacionBurbuja(listaNumeros: List[Int]): List[Int] = {
15     val (nuevaLista, cambio) = pasarBurbuja(listaNumeros)
16     if (!cambio) nuevaLista
17     else ordenacionBurbuja(nuevaLista)
18   }
19
20   def main(args: Array[String]): Unit = {
21     val listaDesordenada = List(19, 8, 12, 3, 10)
22     println(s"Lista original (Immutable): $listaDesordenada")
23     val listaOrdenada = ordenacionBurbuja(listaDesordenada)
24     println(s"Lista ordenada (Funcional): $listaOrdenada")
25   }
26 }
27
```

## Explicaciones del código:

### **Object:**

¿Qué quiere decir exactamente "singleton"? Bueno, en el mundo de la programación, imagina un objeto que vive en la memoria del ordenador, pero solo hay uno, ¡solo uno!, durante todo el tiempo que el programa está funcionando. Solo existe una copia, nada más. Y lo bueno es que podemos acceder a ella directamente, cuando queramos, sin tener que andar creando copias nuevas cada dos por tres.

Este diseño, este "patrón", es muy práctico, sobre todo cuando necesitamos asegurarnos de que algo (una entidad, por ejemplo) tenga el mismo estado en todos lados o, sencillamente, cuando tener varias copias de un objeto no tiene ningún sentido. Personalmente, creo que es una herramienta muy útil.

¿Cómo lo hacemos en Scala?

En Scala, podemos crear un singleton usando la palabra "object". Por ejemplo:

```
object BubbleSortFuncional { ... }
```

Con esto, básicamente, estamos definiendo un objeto llamado BubbleSortFuncional. Sencillo, ¿verdad?

Cosas importantes sobre los singletons en Scala:

No necesitamos usar "new" para crearlo.

En Java, normalmente harías algo como:

```
MiClase obj = new MiClase();
```

Pero en Scala, con solo usar "object", ya podemos acceder a sus métodos directamente:

BubbleSortFuncional.ordenacionBurbuja(lista)

Guarda funciones y valores de forma global.

Todas las funciones y variables que definimos dentro del singleton están disponibles desde cualquier parte del programa, siempre y cuando estemos dentro del "radio de alcance" del objeto.

Es como un contenedor de código.

Puedes imaginar "object" como una especie de "caja" donde metemos funciones y valores que tienen relación entre sí. Así, no tenemos que crear instancias adicionales innecesarias. Esto es muy útil en programas pequeños o cuando queremos hacer algo parecido a las funciones estáticas que usaríamos en Java con "static". Particularmente, me parece que ofrece una buena organización.

### **Función pasarBurbuja:**

En esta función, esencialmente, lo que hacemos es aplicar una iteración, solo una pasada, del conocido algoritmo de burbuja a una lista.

¿El resultado? Pues, lo que obtenemos son dos cosas, dos componentes bien diferenciados.

Primero, una lista nueva. Esta lista contiene exactamente los mismos elementos que la lista original. Sin embargo, aquí está el truco, el valor más grande de la sublista que estamos examinando, como quien dice, ha "burbujeado" hasta el extremo de su alcance. Es como si lo hubiéramos empujado suavemente hacia el final.

Es crucial tener en cuenta que la posición relativa de los demás elementos permanece inalterada. Además, si estamos trabajando con estructuras inmutables, la lista original se mantiene tal cual. En mi opinión, esta característica es muy útil en ciertos contextos de programación.

Y segundo, un indicador booleano, un valor que nos revela si, durante esta pasada que hemos realizado, se ha llevado a cabo al menos un intercambio entre los elementos. En otras palabras, devuelve 'verdadero' si hubo cambios y 'falso' si la sección de la lista ya estaba ordenada. Así sabemos si hemos hecho algo útil o no.

Primero, hablemos del **case**. En Scala, el case es la base del pattern matching. Imagínalo como un switch súper vitaminado. La idea central es simple: "Si los datos se ajustan a este patrón, ¡actúa!". El pattern matching te permite "desarmar" datos de una forma segura y limpia, sin tanto lío de índices y bucles. Facilita mucho la lectura del código, ¿verdad?

Ahora, ¿qué hay de **Nil**? Nil representa una lista vacía, así de sencillo. Es como un `new ArrayList<>()` vacío en Java. Pero aquí viene lo interesante: Nil es un singleton. Esto significa que solo hay una instancia de la lista vacía en toda tu aplicación.

En relación con el case, aparece el símbolo `|`. Este funciona como un "o" lógico. Puedes combinar varios patrones que comparten una misma acción. Por ejemplo: `case Nil | List(_) => (variable, false)`. Esto se traduce como: "Si la variable es Nil o una lista con un solo elemento, entonces devuelve esto otro". El guion bajo `_` actúa como un comodín, perfecto para ignorar el valor específico y enfocarnos en la estructura.

Luego, tenemos `::`, el "cons operator". Este es un poco camaleónico porque sirve para dos cosas: construir y desconstruir listas. Cuando desarmamos una lista con `case x :: y :: tail => ...`, estamos extrayendo el primer elemento (x), el segundo (y), y el resto de la lista (tail). Por otro lado, para construir, simplemente hacemos algo como `val nuevaLista = 0 :: List(1,2,3)`. Añadimos elementos al principio de la lista sin tocar la original.

**Val.** declara una variable inmutable. Una vez que le das un valor, ¡ya no lo puedes cambiar! Es parecido a final en Java. Usar val te ayuda a prevenir errores accidentales y a que tu código sea más predecible, lo que siempre es bienvenido.

Finalmente, el operador `=>` (rocket operator). Dentro del pattern matching, este separa el patrón de la acción. `case patrón => acción` se lee como: "Si la entrada coincide con este patrón, entonces ejecuta esta acción".

Al implementar **bubble sort** de forma funcional, nos alejamos de la modificación directa de la lista original. En cambio, lo que hacemos es simular esos intercambios tan característicos, pero a través de la creación constante de listas nuevas y, por supuesto, llamadas recursivas.

### **Caso 1:** Cuando $x > y$ (Desorden Detectado)

Si detectamos un "fuera de juego", es decir, que el primer elemento ( $x$ ) es mayor que el siguiente ( $y$ ), actuamos de forma diferente.

En lugar del típico "swap" en el mismo lugar, preferimos poner  $y$  (el más pequeño) al frente de la fila, por así decirlo.

Después, llamamos a la función de nuevo, pero esta vez sobre la sublista que comienza con  $x$  y el resto de la cola (tail):

```
pasarBurbuja(x :: tail)
```

La idea detrás de  $x :: tail$  es que  $x$  todavía tiene que "medirse" con el siguiente elemento, así que lo dejamos en espera para la siguiente ronda recursiva.

Tomamos la lista que resulta de esta recursión (rest) y la devolvemos como una lista flamante, construida así:

```
(y :: rest, true)
```

Ese "true" ahí es un indicador de que, efectivamente, hubo un intercambio durante esta pasada.

### **Caso 2:** $x \leq y$ (Todo en Orden)

Si  $x$  e  $y$  están ya en su sitio, simplemente colocamos  $x$  al frente de la lista.

Iniciamos una llamada recursiva sobre la sublista que comienza con  $y :: tail$ , porque los siguientes elementos también necesitan ser evaluados.



Finalmente, retornamos:

(x :: rest, swapped)

Aquí, swapped es el valor que viene "de abajo", desde la recursión.

Esto nos da la capacidad de transmitir la información sobre si hubo algún intercambio en algún punto de la lista, para que, al final, sepamos con certeza si la pasada introdujo algún cambio o no. En mi opinión, esta aproximación funcional, aunque quizás más compleja al principio, ofrece una elegancia y claridad conceptual notables.

### **Función ordenacionBurbuja:**

En la versión funcional del ordenamiento burbuja, en lugar de los bucles de toda la vida, echamos mano de la recursión para manejar esas pasadas repetidas sobre la lista hasta que, queda completamente ordenada. Digamos que es darle una vuelta de tuerca al concepto.

Primero, llamamos a la función pasarBurbuja con la lista que tengamos entre manos. Esta función nos devuelve un par de cositas:

nuevaLista: que es la lista resultante tras darle una pasada.

cambio: un booleano que nos chiva si hubo al menos un intercambio durante esa pasada (true si se movió algo, false si todo quedó igual).

Ahora, toca evaluar ese "cambio".

Si "cambio" es false, la lista ya está en su sitio. No hubo intercambios, así que devolvemos nuevaLista como resultado final y a otra cosa. En mi opinión, esta es la parte más elegante.

Pero si "cambio" es true, ojo, que aún hay elementos fuera de lugar. Toca repetir la jugada, llamando recursivamente a la función con nuevaLista.

Así, paso a paso, conseguimos ordenar la lista sin tocar la original, respetando la inmutabilidad:

Cada pasada crea una nueva lista, sin alterar la anterior, que queda ahí intacta.

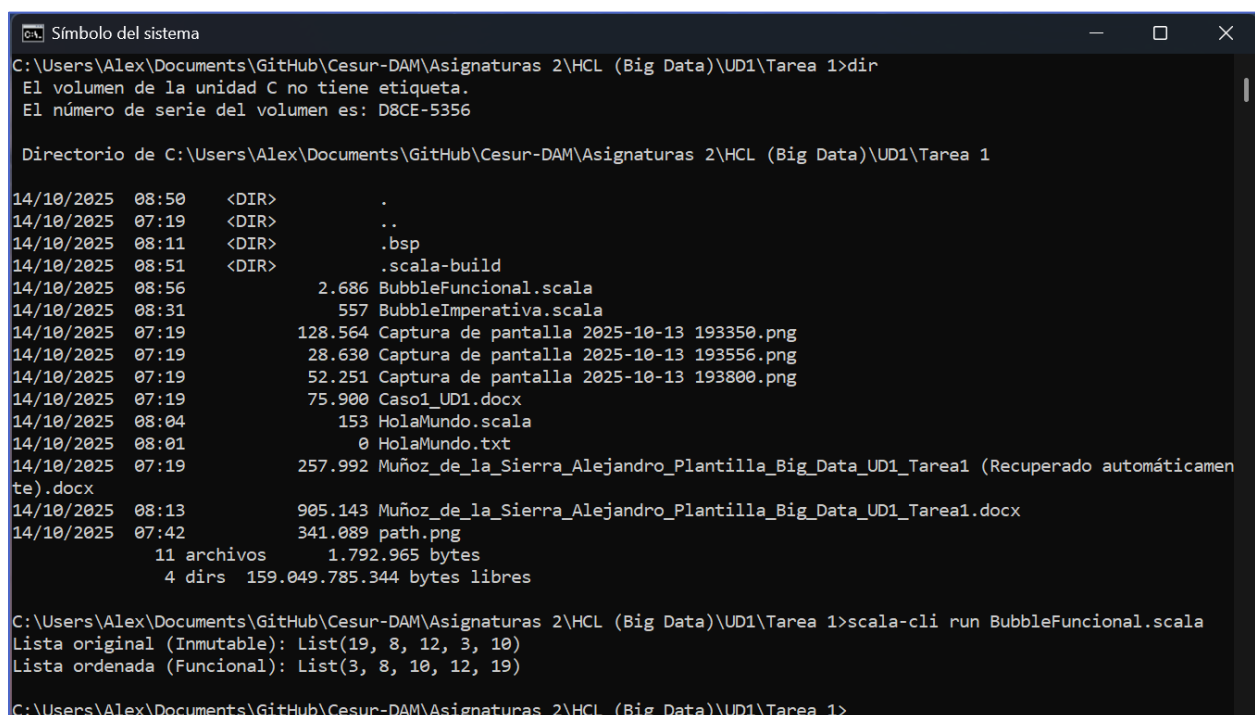
La recursión hace el papel del bucle, controlando cuántas pasadas son necesarias hasta que la lista esté perfecta.

Y el valor "cambio" nos chiva cuándo parar, diciéndonos exactamente cuándo dejar de dar pasadas.

Este enfoque, en resumen, muestra cómo los algoritmos clásicos pueden adaptarse al estilo funcional, combinando recursión, inmutabilidad y propagación de estado. Es una forma interesante de ver las cosas, y yo diría que bastante ingeniosa.

Por último la Función **Main**:

Imprimimos la lista original , llamamos a la función de ordenación e imprimimos el resultado:



```
Símbolo del sistema
C:\Users\Alex\Documents\GitHub\Cesur-DAM\Asignaturas 2\HCL (Big Data)\UD1\Tarea 1>dir
El volumen de la unidad C no tiene etiqueta.
El número de serie del volumen es: D8CE-5356

Directorio de C:\Users\Alex\Documents\GitHub\Cesur-DAM\Asignaturas 2\HCL (Big Data)\UD1\Tarea 1
14/10/2025 08:50 <DIR>      .
14/10/2025 07:19 <DIR>      ..
14/10/2025 08:11 <DIR>      .bsp
14/10/2025 08:51 <DIR>      .scala-build
14/10/2025 08:56      2.686 BubbleFuncional.scala
14/10/2025 08:31      557 BubbleImperativa.scala
14/10/2025 07:19    128.564 Captura de pantalla 2025-10-13 193350.png
14/10/2025 07:19    28.630 Captura de pantalla 2025-10-13 193556.png
14/10/2025 07:19    52.251 Captura de pantalla 2025-10-13 193800.png
14/10/2025 07:19    75.900 Caso1_UD1.docx
14/10/2025 08:04    153  HolaMundo.scala
14/10/2025 08:01      0  HolaMundo.txt
14/10/2025 07:19   257.992 Muñoz_de_la_Sierra_Alejandro_Plantilla_Big_Data_UD1_Tarea1 (Recuperado automáticamente).docx
14/10/2025 08:13    905.143 Muñoz_de_la_Sierra_Alejandro_Plantilla_Big_Data_UD1_Tarea1.docx
14/10/2025 07:42    341.089 path.png
                11 archivos      1.792.965 bytes
                4 dirs  159.049.785.344 bytes libres

C:\Users\Alex\Documents\GitHub\Cesur-DAM\Asignaturas 2\HCL (Big Data)\UD1\Tarea 1>scala-cli run BubbleFuncional.scala
Lista original (Immutable): List(19, 8, 12, 3, 10)
Lista ordenada (Funcional): List(3, 8, 10, 12, 19)

C:\Users\Alex\Documents\GitHub\Cesur-DAM\Asignaturas 2\HCL (Big Data)\UD1\Tarea 1>
```

## CONCLUSIONES

En esta práctica, he tenido la oportunidad de sumergirme en la sintaxis de Scala, explorando elementos cruciales como `object`, `def`, `val`, las listas inmutables (¡tan útiles!), el pattern matching y las funciones recursivas. Personalmente, me ha parecido fascinante cómo Scala logra fusionar la programación funcional y orientada a objetos, dándonos la flexibilidad de encarar los problemas desde distintas perspectivas, según lo que mejor se adapte a la situación.

Programación funcional:

La implementación del famoso algoritmo de burbuja me reveló que es factible abordar problemas conocidos sin alterar los datos originales. Usando la recursión y creando nuevas listas en cada etapa, se hace evidente que fomentamos la inmutabilidad y la noción de funciones puras – donde cada función produce resultados predecibles y sin "sorpresas" indeseadas.

Pattern matching y listas:

Conceptos como `::`, `Nil`, `case` y `_` nos dan la capacidad de "desarmar" y reconstruir listas de una manera que, a mi parecer, es excepcionalmente clara y elegante. Gracias a ellos, el código se vuelve más legible y expresivo, mostrando de manera explícita la estructura de los datos y las operaciones que les aplicamos. A mí, me parece que esto marca la diferencia en la mantenibilidad del código.

Recursión y control de flujo:

Con funciones como `pasarBurbuja` y `ordenacionBurbuja`, hemos logrado entender cómo manejar iteraciones a través de la recursión, dejando atrás los bucles tradicionales. Además, las tuplas nos permiten devolver varios valores de forma ordenada, como la lista parcial y un indicador de si hubo algún intercambio. Esto, en mi opinión, facilita un control de flujo más preciso y seguro.

## R E F E R E N C I A S

<https://docs.scala-lang.org/>

<https://docs.scala-lang.org/scala3/book/fp-intro.html>

<https://www.baeldung.com/scala/functional-programming>

<https://medium.com/%40abhishekranjandev/from-zero-to-hero-unmasking-functional-programming-in-scala-de64641a12d9>

<https://docs.scala-lang.org/tour/pattern-matching.htm>

<https://www.baeldung.com/scala/pattern-matching>

<https://runestone.academy/ns/books/published/pythoned/SortSearch/ElOrdenamientoBurbuja.html>