**A blue and black logo

Description automatically generated**

**Programming Paradigms:**

**Coursework**

**Evaluation and Testing**

**2024-2025**

**Department of Computing**

**Submitted for the Degree of**:

BSc Software Development for Business

**Name:** Vincenzo Russo

**Programme :** Software Development for Business

**Matriculation Number:** S2336012

**Word Count:** 1712

**(*excluding contents pages, figures, tables, references and Appendices*)**

**“Except where explicitly stated, all work in this report, is my own original work and has not been submitted elsewhere in fulfilment of the requirement of this or any other award”**



**Signed by Student:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date:** 11/12/2024



Table of Contents

[1. Evaluation 4](#_Toc184843239)

1.2 Project Background……………………………………………………………………4

[2. Functional Thinking 5](#_Toc184843240)

[2.1 Immutability 5](#_Toc184843241)

[2.2 Pure Functions 5](#_Toc184843242)

[3. Functional Techniques 6](#_Toc184843243)

[3.1 Higher-Order Functions 6](#_Toc184843244)

[3.2 Tail Recursion 6](#_Toc184843245)

[3.3 Option, Try, Either and Pattern Matching 7](#_Toc184843246)

[4. Functional vs. Imperative Approaches 8](#_Toc184843247)

[4.1 Functional Approach 8](#_Toc184843248)

[4.2 Alternative Imperative Approach 8](#_Toc184843249)

[4.3 Conclusion 8](#_Toc184843250)

[5. Test Log 9](#_Toc184843251)

[Appendices 13](#_Toc184843252)

[Appendix A – Evaluation Large Code Screenshots 13](#_Toc184843253)

[Appendix B – Tables 17](#_Toc184843255)

[Appendix C – Testing Screenshots: 18](#_Toc184843256)

[References 27](#_Toc184843257)

Table of Figures

[Figure 1: mapData 5](#_Toc184843269)

[Figure 2: getSelectedPoints 6](#_Toc184843270)

[Figure 3: getAvgPoints Function 13](#_Toc184843271)

[Figure 4: handleInput Function 14](#_Toc184843272)

[Figure 5: Partially Applied Curried Functions 14](#_Toc184843273)

[Figure 6: readFile Function 15](#_Toc184843274)

[Figure 7: menuLoop Function 16](#_Toc184843275)

[Figure 8 18](#_Toc184843276)

[Figure 9 19](#_Toc184843277)

[Figure 10 20](#_Toc184843278)

[Figure 11 21](#_Toc184843279)

[Figure 12 21](#_Toc184843280)

[Figure 13 22](#_Toc184843281)

[Figure 14 22](#_Toc184843282)

[Figure 15 23](#_Toc184843283)

[Figure 16 23](#_Toc184843284)

[Figure 17 24](#_Toc184843285)

[Figure 18 24](#_Toc184843286)

[Figure 19 24](#_Toc184843287)

[Figure 20 25](#_Toc184843288)

[Figure 21 25](#_Toc184843289)

[Figure 22 25](#_Toc184843290)

[Figure 23 26](#_Toc184843291)

[Figure 24 26](#_Toc184843292)

[Figure 25 26](#_Toc184843293)

Table of Tables

[Table 1: Testing Table 12](#_Toc184843802)

[Table 2: Tail Recursion Example 17](#_Toc184843803)

# Evaluation

## Project Background

This project involved creating a Scala Singleton-object-based console application designed to perform statistical analysis on Formula One data. The console acts a User-Interface (UI) for the end-user, accepting a numerical input for analysis selection. The application reads data from a text file and displays six statistical summaries to the end-user. Data is then parsed into a Map, where the season year ([Int]) serves as the key, and its corresponding value is a List of Tuples containing the driver’s name ([String]), points ([Float]), and wins ([Int]). IntelliJ IDEA was chosen as the development environment due to its comprehensive support for JVM languages like Scala, providing advanced tools for efficient code development and debugging (Hunt, 2014, p. 479; JetBrains, 2024).

# Functional Thinking

Functional thinking focusses on declaring *what* a program should do, rather than *how* it should do it. This contrasts imperative thinking, where an application instead details a list of instructions it should follow. Central concepts that inspired functional ideals in the development process were immutability and pure functions.

## Immutability

In Scala, variable declaration revolves around two dynamic and statically typed identifiers: val and var. The latter is a mutable identifier that allows for value reassignment after initialisation at runtime. In contrast, the former val cannot be changed once assigned without flagging a compilation error; it’s an immutable-typed variable, which is good in this context.

Immutability promoted concurrency and thread-safety when programming the application; it allowed for safe sharing of data across different threads at runtime by reusing existing structures. The approach reduced computational overhead by encouraging the use of optimal algorithms and memory management practices, thereby minimising memory misuse.

A computer screen shot of a program code

Description automatically generated

**Figure 1: mapData**

Figure 1 above exemplifies this, located in the global scope of the application. Here, mapData is defined using val and assigned to an immutable collection structure, Map, where the data is collected from an alternate function. The immediacy of the dataset declared immutable forced subsequent functional thinking throughout development, focussing on transformation rather than mutation. In contrast, an imperative approach would rely on mutable state, increasing the risk of race conditions and inefficiencies due to unnecessary memory allocations. Instead of altering mapData directly, pure functions were employed to operate on its contents, promoting safer, predictable code with a declarative style.

## Pure Functions

Pure functions produce the same output for any given input, i.e., create no side-effects. In the application, pure backend functions were designed to express the intended results declaratively by describing *what* should be computed rather than specifying *how* to compute it. This approach allowed a clear separation of concerns by consistently returning new Maps based on selected analysis criteria.

For example, as seen in Appendix A, Figure 3, the function getAvgPoints, when given the same input data Map, declaratively returns the same output of all drivers’ average points across all seasons. This ensures referential transparency; the program’s expected behaviour remains unchanged, and execution stays consistent. The function’s consistency preserves data integrity, especially in floating-point calculations. Slight variations in input point values could significantly alter the average point calculation – undesirable since accuracy is essential.

# Functional Techniques

Efforts were made to achieve an application of high standard using available multiple techniques. Not all techniques are discussed; however, the following section will explore notable ones.

## Higher-Order Functions

A higher-order function is a function that either takes one or more functions as arguments, or returns a function as a result. In the application, higher-order functions were used extensively to abstractly derive results, enabling clear and reusable logic without explicitly manipulating the dataset’s values. The ability to chain them was also a large component in development, allowing to perform complex transformations in a clean, modular way.A screenshot of a computer program

Description automatically generated

**Figure 2: getSelectedPoints**

In Figure 2 above, the getSelectedPoints function demonstrates this use of higher-order functions. This backend function takes data as a parameter and applies three succinct higher-order transformations to filter for the driver that matches the end-user’s selection. It then sums up all their points per season.

**Each Function’s Role:**

* **.collect:** Initiates the chain of operations. It traverses the data map and applies a partial function to extract relevant driver lists from each season. It processes only the entries matching the specified pattern and discards others.
* **.filter:** Is applied to the list of drivers to filter out entries that match a specific criterion. The filtering logic uses a pattern match where a function compares the driver’s name to the specified fullName ignoring case.
* **.map:** After filtering, .map is used to extract the points of each matching driver using pattern matching. This creates a list of points corresponding to the selected driver across multiple seasons.

This approach highlights the power of combining higher-order functions to perform complex data transformations concisely and declaratively, without relying on traditional loops. While a foreach loop could have been used for iteration, the focus was to explore as many different functional looping techniques as possible.

## Tail Recursion

Tail recursion is a special form of recursion where a function calls itself as its last operation before returning a result. This allows the compiler to optimise the call, reusing the same stack-frame instead of creating new ones for each recursion. The application uses tail recursion in a few instances.

In Appendix A, Figure 3, the recursive call calculateAvg(tail, sumFloat(total, points), count + 1) is the last operation performed in each recursive step. Since there’s nothing left to process after this last call, the compiler can reuse the same stack frame, making the recursion memory-efficient through tail-call optimisation. The compiler treats this function as tail-recursive through the @tailrec annotation.

As shown in Appendix B, Table 2, after each recursive call, the same stack frame is reused, and the points are updated until the base case is reached. In the base case, where the list is empty (Nil), the total points (2458) are divided by the count of drivers (22), resulting in the average of 111.73. No new stack frame is created during the recursion, unlike traditional recursion, where each call would add a new stack frame – leading to the risk of a stack overflow. This is why tail recursion was chosen over traditional recursion.

## Option, Try, Either and Pattern Matching

Option, Try, Either and Pattern Matching are functional ways to handle the existence/absence of elements, error handling and finding patterns in code respectively. Instead of imperative ways, like traditional if/else statements and throwing stack exceptions, these methods were used throughout the application to handle unexpected behaviours.

This is especially true as seen in Appendix A, Figure 4. The handleInput function uses all three of these to divert the application to the appropriate function based on whether or not end-user input is valid. The function is also curried to allow for partial application. This means the function can be used with different inputs with different variable types, passed when called/needed, disabling the need to rewrite the logic for input handling each time (See Appendix A, Figure 5).

**Each Technique’s Role:**

* **Option and Try**: The function uses Try to attempt to convert the input string to an integer. It’s then converted to Option type. If the conversion fails (i.e., the input is a name, and can’t be converted to an integer), it returns None, representing the absence of a valid value. This is then handled appropriately by the function.
* **Either:** The function returns an either value, where the Left side contains an error message in case of invalid input, and the Right side contains the valid input (either an integer or a string), depending on whether the input can be successfully processed.
* **Pattern Matching:** The function uses pattern matching to distinguish between the different possible outcomes of Try, Option, and Either. It matches on the Some or None result from Try, and on the Right or Left result from either, allowing the application to follow different execution paths based on the validity of the input.

This approach condenses complex logic into a single function, where otherwise separate functions would be needed to handle different types of input from the command-line menu. The function is reusable based on different contexts, directing the end-user to the function or error based on the call’s location. Alternatively, for-comprehensions could’ve been used, though at time of programming was unfamiliar. They work by combining multiple operations in a sequential style, automatically handling failures and None values.

# Functional vs. Imperative Approaches

## Functional Approach

Initially, Scala was challenging due to its declarative nature. However, gaining a deeper understanding of concepts like functional loops and immutability in the readFile function led to a more efficient approach. Using foldLeft and immutable variables throughout the function made data processing concise and seamless, eliminating the need for explicit state management. Error handling with Try reduced clutter and enhanced debugging by capturing file-related issues early (See Appendix A, Figure 6). Tail recursion also replaced traditional looping structures like do-while, optimising performance by reusing stack frames. This technique not only streamlined the menu implementation but also proved reusable across other project components (See Appendix A, Figures 3 and 7).

## Alternative Imperative Approach

Coming from an imperative programming background, C# would have been a more familiar choice for developing this application. In such an approach, a dedicated DataManager class would encapsulate file-reading logic using File.ReadAllLines, with conditional if statements to validate and parse each line of the dataset. A menu would be created using a do-while loop, allowing users to interact with the program. However, reliance on mutable variables and nested conditional logic could complicate debugging and reduce readability compared to a functional approach using immutable data structures.

## Conclusion

In retrospect, while Scala required a mindset shift from imperative to functional programming, its capabilities for readability and maintainability proved undeniable. Given the opportunity to choose any language for the application requirements, Scala would now be the preferred choice. Ultimately, the journey from initial challenges to gaining a deeper understanding of its functional paradigm highlighted Scala’s value as a powerful tool, encouraging the adoption of a functional approach in future projects.

# Test Log

The following section contains the test cases for the Formula 1 console application written in Scala. Each test includes the respective test number, any input or actions needed, the expected output, actual output, result and screenshot reference number – included in the appendices. See next page and Appendix C.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test No.** | **Input/Action** | **Expected Output** | **Actual Output** | **Result** | **Screenshot Reference**  **(Appendix C)** |
| **1** | Start application. | Console welcomes user, prompts user to input number 1-7. | As expected. | **Pass** | Figure 8 |
| **2** | Start application. Input ‘1’, hit enter. | Console prints “Option 1 selected…”. Console then prints all season winners with their points and wins for that year, then loops back to menu prompting another selection. | As expected. | **Pass** | Figure 9 |
| **3** | Start application. Input ‘2’, hit enter. Input ‘2017’, hit enter. | Console prints “Option 2 selected…”. Console prints “Please enter the season you want to display:”. Console prints all statistics of each driver for that season inputted, then loops back to menu prompting another selection. | As expected. | **Pass** | Figure 10 |
| **4.1** | Start application. Input ‘2’, hit enter. Input ‘s’, hit enter. | Console prints “ Invalid input 's'. Please enter a valid year between 2017 and 2023.”, then loops back to menu prompting another selection. | As expected. | **Pass** | Figure 11 |
| **4.2** | Start application. Input ‘2’, hit enter. Input ‘2016’ (invalid year), hit enter. | Console prints “Invalid input '2016'. Please enter a valid year between 2017 and 2023.”, then loops back to menu prompting another selection | As expected. | **Pass** | Figure 12 |
| **5** | Start application. Input ‘3’, hit enter. | Console prints “Option 3 selected…”. Console prints total wins per season, then loops back to menu prompting another selection. | As expected. | **Pass** | Figure 13 |
| **6** | Start application. Input ‘4’, hit enter. | Console prints “Option 4 selected…”. Console prints average points per season rounded up, then loops back to menu prompting another selection. | As expected. | **Pass** | Figure 14 |
| **7** | Start application. Input ‘5’, hit enter. | Console prints “Option 5 selected…”. Console prints total points per season ascending, then loops back to menu prompting another selection. | As expected. | **Pass** | Figure 15 |
| **8.1** | Start application. Input ‘6’, hit enter. Input ‘Lewis Hamilton’, hit enter. | Console prints “Option 6 selected…” Console prints “Please enter the player you want to display (case sensitive):”. Console prints driver and their total points, then loops back to menu prompting another selection. | As expected | **Pass** | Figure 16 |
| **8.2.** | Start application. Input ‘6’, hit enter. Input ‘2’, hit enter. | Console prints “Invalid input ‘2’. The name wasn’t found. Please enter a valid name from the list…”, then loops back to menu prompting another selection. | As expected. | **Pass** | Figure 17 |
| **8.2** | Start application. Input ‘6’, hit enter. Input name with wrong case last name, e.g., ‘Lewis hamilton’, hit enter. | Console prints driver and their total points, then loops back to menu prompting another selection. | As expected. | **Pass** | Figure 18 |
| **8.4** | Start application. Input ‘6’, hit enter. Hit enter. | Console prints “Invalid input ‘’. The name wasn’t found. Please enter a valid name from the list…”, then loops back to menu prompting another selection. | As expected. | **Pass** | Figure 19 |
| **8.5** | Start application. Input ‘6’, hit enter. Input name with three names, e.g., ‘Paul di Resta’. Hit enter. | Console prints driver and their total points, then loops back to menu prompting another selection. | As expected. | **Pass** | Figure 20 |
| **8.6** | Start application. Input ‘6’, hit enter. Input name with three names with wrong case in middle name, e.g., ‘Paul Di Resta’. Hit enter. | Console prints driver and their total points, then loops back to menu prompting another selection. | As expected. | **Pass** | Figure 21 |
| **8.7** | Start application. Input ‘6’, hit enter. Input name with inappropriate spaces, e.g., ‘Lewis Hamilton’. | Console prints driver and their total points, then loops back to menu prompting another selection. | As expected. | **Pass** | Figure 22 |
| **9.1** | Start application Input ‘7’, hit enter. | Console prints “Option 7 selected…” Console prints “Quitting the application…” Application quits. | As expected. | **Pass** | Figure 23 |
| **9.2** | Start application with no ‘data.txt’ file present in filepath ‘./src/resources/data.txt’. | Console prints “Error reading file: ./src/resources/data.txt (No such file or directory)”. Application quits. | As expected. | **Pass** | Figure 24 |
| **9.3** | Start application with empty data.txt file. | Console prints “Error: The data file is empty.” Application quits. | As expected. | **Pass** | Figure 25 |

**Table 1: Testing Table**

# Appendices

## Appendix A – Evaluation Large Code Screenshots

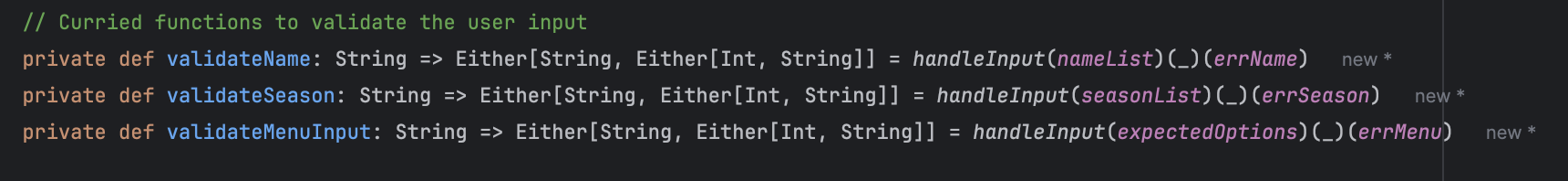
## 

**Figure 3: getAvgPoints Function**

A computer screen with green and white text

Description automatically generated

**Figure 4: handleInput Function**



**Figure 5: Partially Applied Curried Functions**

A screenshot of a computer program

Description automatically generated

**Figure 6: readFile Function**

A screen shot of a computer program

Description automatically generated

**Figure 7: menuLoop Function**

## Appendix B – Tables

For example, consider the data for year 2023:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Call # | Current Drivers | Total Points (Accumulated) | Count | Stack Frame Status |
| 1 | (“Max Verstappen”, 575, 19) … | 0 + 575 = 575 | 0 + 1 = 1 | Stack frame created; points updated |
| 2 | (“Sergio Perez”, 285, 2) … | 575 + 285 = 860 | 1 + 1 = 2 | Same frame reused; points updated |
| 3 | (“Lewis Hamilton”, 234, 0) … | 860 + 234 = 1094 | 2 + 1 = 3 | Same frame reused; points updated |
| … | … | … | … | … |
| 22 | (“Nyck de Vries”, 0, 0), Nil | 2458 + 0 = 2458 | 21 + 1 = 22 | Same frame reused; points updated |
| 23 | Nil (Base case reached) | 2458 | 22 | Base case met. Average = 2458 / 22 = 111.73 |

**Table 2: Tail Recursion Exam****ple**

## Appendix C – Testing Screenshots:

**A screenshot of a computer

Description automatically generated**

**Figure 8**

A screenshot of a computer

Description automatically generated

**Figure 9**

A screenshot of a computer

Description automatically generated

**Figure 10**

A screenshot of a computer program

Description automatically generated

**Figure 11**

**A screenshot of a computer program

Description automatically generated**

**Figure 12**

A screenshot of a computer

Description automatically generated

**Figure 13**

**A screenshot of a computer

Description automatically generated**

**Figure 14**

A screenshot of a computer

Description automatically generated

**Figure 15**

**A screenshot of a computer program

Description automatically generated**

**Figure 16**

A screenshot of a computer program

Description automatically generated

**Figure 17**

**A screenshot of a computer program

Description automatically generated**

**Figure 18**

A screenshot of a computer program

Description automatically generated

**Figure 19**

A screenshot of a computer program

Description automatically generated

**Figure 20**

A screenshot of a computer program

Description automatically generated

**Figure 21**

A screenshot of a computer program

Description automatically generated

**Figure 22**

A screenshot of a computer program

Description automatically generated

**Figure 23**

**A screenshot of a computer

Description automatically generated**

**Figure 24**

A screenshot of a computer

Description automatically generated

**Figure 25**

# References

Hunt, J. (2014). *A Beginner's Guide to Scala, Object Orientation and Functional Programming*. Cham: Springer

JetBrains. (2024). *Advanced configuration*. Available at: <https://www.jetbrains.com/help/idea/tuning-the-ide.html> (Accessed: 7 December 2024).