



**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
1.2 Project Background.....	4
2. Functional Thinking .....	5
2.1 Immutability.....	5
2.2 Pure Functions .....	5
3. Functional Techniques .....	6
3.1 Higher-Order Functions .....	6
3.2 Tail Recursion .....	6
3.3 Option, Try, Either and Pattern Matching .....	7
4. Functional vs. Imperative Approaches .....	8
4.1 Functional Approach.....	8
4.2 Alternative Imperative Approach .....	8
4.3 Conclusion .....	8
5. Test Log .....	9
Appendices.....	13
Appendix A – Evaluation Large Code Screenshots.....	13
Appendix B – Tables .....	17
Appendix C – Testing Screenshots: .....	18
References.....	27

## Table of Figures

Figure 1: mapData.....	5
Figure 2: getSelectedPoints.....	6
Figure 3: getAvgPoints Function .....	13
Figure 4: handleInput Function.....	14
Figure 5: Partially Applied Curried Functions.....	14
Figure 6: readFile Function.....	15
Figure 7: menuLoop Function .....	16
Figure 8 .....	18
Figure 9 .....	19
Figure 10 .....	20
Figure 11 .....	21
Figure 12 .....	21
Figure 13 .....	22
Figure 14 .....	22
Figure 15 .....	23
Figure 16 .....	23
Figure 17 .....	24
Figure 18 .....	24
Figure 19 .....	24
Figure 20 .....	25
Figure 21 .....	25
Figure 22 .....	25
Figure 23 .....	26
Figure 24 .....	26
Figure 25 .....	26

## Table of Tables

Table 1: Testing Table .....	12
Table 2: Tail Recursion Example .....	17

# 1. Evaluation

## 1.1 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 as 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).

## 2. 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.

### 2.1 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.

```
// Read data from file via readFile utility function and store in map
private val mapData: Map[Int, List[(String, Float, Int)]] = readFile(dataFilePath) match {
  case Right(data) => data // Return the data if successful
  case Left(error) => // Handle the error
    println(s"Error reading data file: $error")
    sys.exit(1)
}
```

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.

### 2.2 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.

## 3. 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.

### 3.1 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.

```
// Backend function to get selected driver's total points
private def getSelectedPoints(data: Map[Int, List[(String, Float, Int)]], inputName: String): Map[String, Float] = {
  val fullName = inputName
  val sumPoints = data.collect { case (_, drivers) =>
    // Filter based on simple function to check if the name matches
    drivers.filter { case (name, _, _) => name.equalsIgnoreCase(fullName) }
    // Sum selected driver
    .map { case (_, points, _) => points }.sum
    // Sum all seasons
  }.sum

  // Return the selected driver's total points as a Map
  Map(fullName -> sumPoints)
}
```

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.

### 3.2 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.

### 3.3 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.

## **4. Functional vs. Imperative Approaches**

### **4.1 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).

### **4.2 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.

### **4.3 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.



## 5. 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

<b>9.1</b>	Start application Input '7', hit enter.	Console prints "Option 7 selected..." Console prints "Quitting the application..." Application quits.	As expected.	<b>Pass</b>	Figure 23
<b>9.2</b>	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.	<b>Pass</b>	Figure 24
<b>9.3</b>	Start application with empty data.txt file.	Console prints "Error: The data file is empty." Application quits.	As expected.	<b>Pass</b>	Figure 25

**Table 1: Testing Table**

# Appendices

## Appendix A – Evaluation Large Code Screenshots

```
// Backend function to get average points
private def getAvgPoints(data: Map[Int, List[(String, Float, Int)]]): Map[Int, Float] = {  ⚡ vrusso300

  // Calculate the average points for each season in a tail-recursive func
  @tailrec
  def calculateAvg(drivers: List[(String, Float, Int)], total: Float, count: Int): Float = drivers match {
    // Base case, if the list is empty, return the average
    case Nil => if (count == 0) 0 else total / count
    // Recursive case, sum the points via external sumFloat func and increment the count
    case (_, points, _) :: tail => calculateAvg(tail, sumFloat(total, points), count + 1)
  }

  // Map the seasons to the average points
  val result: Map[Int, Float] = data.map { case (season, drivers) =>
    // Calculate the average points for each season
    val avgPoints = calculateAvg(drivers, 0, 0)
    // Round the points up to 2 decimal places using big decimal, then convert to float
    val roundedAverage = BigDecimal(avgPoints).setScale(2, BigDecimal.RoundingMode.HALF_UP).toFloat
    season -> roundedAverage
  }

  // Return the average points for seasons
  result
}
```

Figure 3: getAvgPoints Function

```
// Curried Function to validate and process the input based on a list of expected values
private def handleInput(validOptions: List[Any])(input: String)(contextMessage: String): Either[String, Either[Int, String]] = ⌚ vrusso300 *
  Try(input.toInt).toOption match {
    case Some(option) if validOptions.contains(option) =>
      // If parse is successful and the option is in the list, return the option
      Right(Left(option))

    // If parse fails (i.e., is a name and cant be an integer), check if its in the passed list
    case None if validOptions.map(_.toString.toLowerCase).contains(input.toLowerCase) =>
      Right(Right(input))
    case _ =>
      // Return the error message if the input is invalid
      Left(s"Invalid input '$input'. $contextMessage")
  }
}
```

Figure 4: handleInput Function

```
// Curried functions to validate the user input
private def validateName: String => Either[String, Either[Int, String]] = handleInput(nameList)(_)(errName) new *
private def validateSeason: String => Either[String, Either[Int, String]] = handleInput(seasonList)(_)(errSeason) new *
private def validateMenuInput: String => Either[String, Either[Int, String]] = handleInput(expectedOptions)(_)(errMenu) new *
```

Figure 5: Partially Applied Curried Functions

```

private def readFile(fileName: String): Either[String, Map[Int, List[(String, Float, Int)]]] = {  ± vrusso300
  // Safely manage file reading
  Using(Source.fromFile(fileName)) { bufferedSource =>
    val lines = bufferedSource.getLines().toList

    val data = lines.headOption match {
      case None => return Left("File is empty")
      case Some(_) =>
        // Use foldLeft to accumulate the result in an immutable map
        lines.foldLeft(Map[Int, List[(String, Float, Int)]>()) {
          case (mapData, line: String) =>
            val splitLine = line.split(",", 2).map(_.trim)
            val season = splitLine(0).toInt
            val allDrivers = splitLine(1).split(";").map(_.trim)

            // Process each driver entry and create a list of tuples
            val seasonData = allDrivers.map { driverEntry =>
              val driverDetails = driverEntry.split(":").map(_.trim)
              val name = driverDetails(0)
              val stats = driverDetails(1).split(" ").map(_.trim)
              val points = stats(0).toFloat
              val wins = stats(1).toInt
              (name, points, wins)
            }.toList

            // Update the map with the new season data, appending to the existing list
            mapData + (season -> (mapData.getOrElse(season, List()) ++ seasonData))
          }
        }
      Right(data)
    } match {
      case Success(data) => println("Data loaded successfully"); data
      case Failure(exception) => Left(exception.getMessage)
    }
  }
}

```

Figure 6: readFile Function

```
private def menuLoop(): Unit = {   vrusso300
  // Tailrec loop
  @tailrec
  def loop(): Unit = {
    val input = displayMenuAndReadOption()
    validateMenuInput(input) match {
      case Right(Left(option)) =>
        if (processMenuOption(option)) loop()
      case Left(error) =>
        println(error)
        loop()
    }
  }
  // Start the recursive loop
  loop()
}
```

Figure 7: menuLoop Function



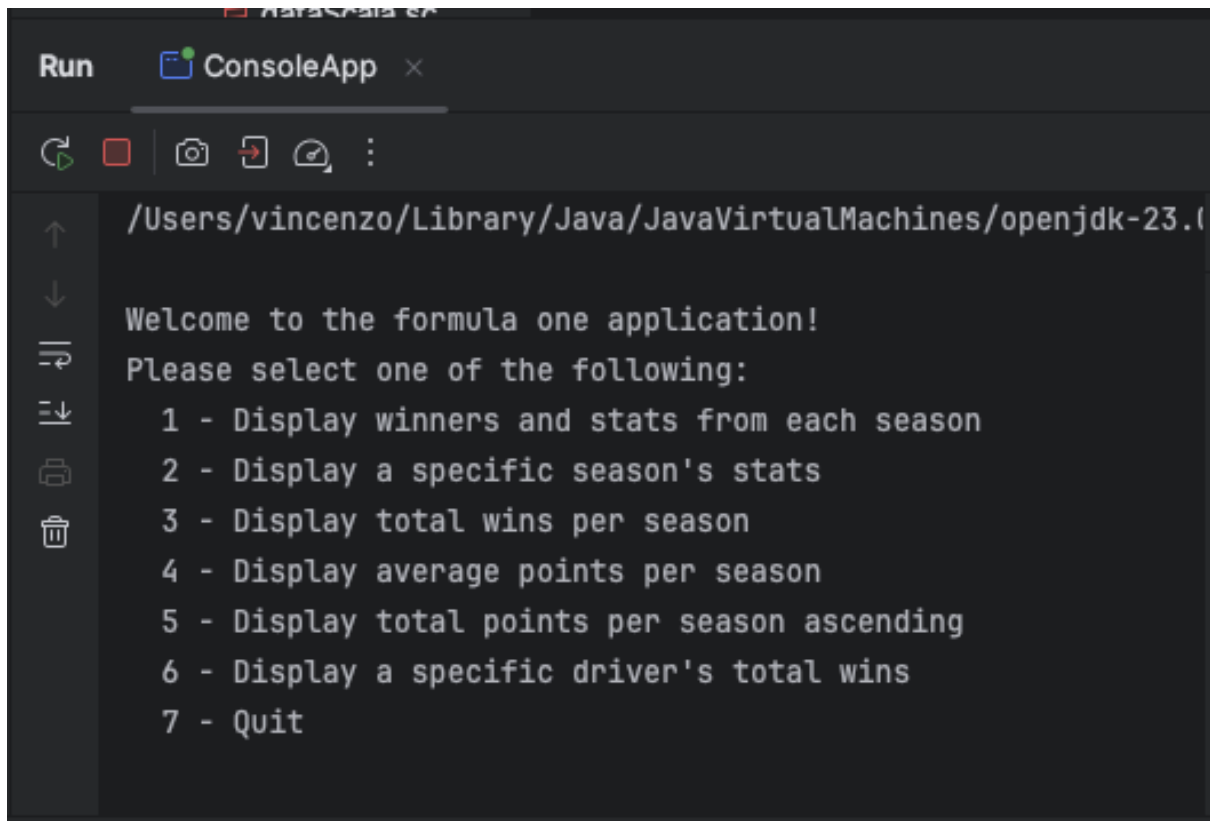
## Appendix B – Tables

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 Example

## Appendix C – Testing Screenshots:

A screenshot of an IDE's console window. The title bar shows 'Run' and 'ConsoleApp'. The console output displays the path '/Users/vincenzo/Library/Java/JavaVirtualMachines/openjdk-23.0', followed by a welcome message and a list of seven menu options for a formula one application. The options are: 1 - Display winners and stats from each season, 2 - Display a specific season's stats, 3 - Display total wins per season, 4 - Display average points per season, 5 - Display total points per season ascending, 6 - Display a specific driver's total wins, and 7 - Quit.

```
Run ConsoleApp x
/Users/vincenzo/Library/Java/JavaVirtualMachines/openjdk-23.0
Welcome to the formula one application!
Please select one of the following:
1 - Display winners and stats from each season
2 - Display a specific season's stats
3 - Display total wins per season
4 - Display average points per season
5 - Display total points per season ascending
6 - Display a specific driver's total wins
7 - Quit
```

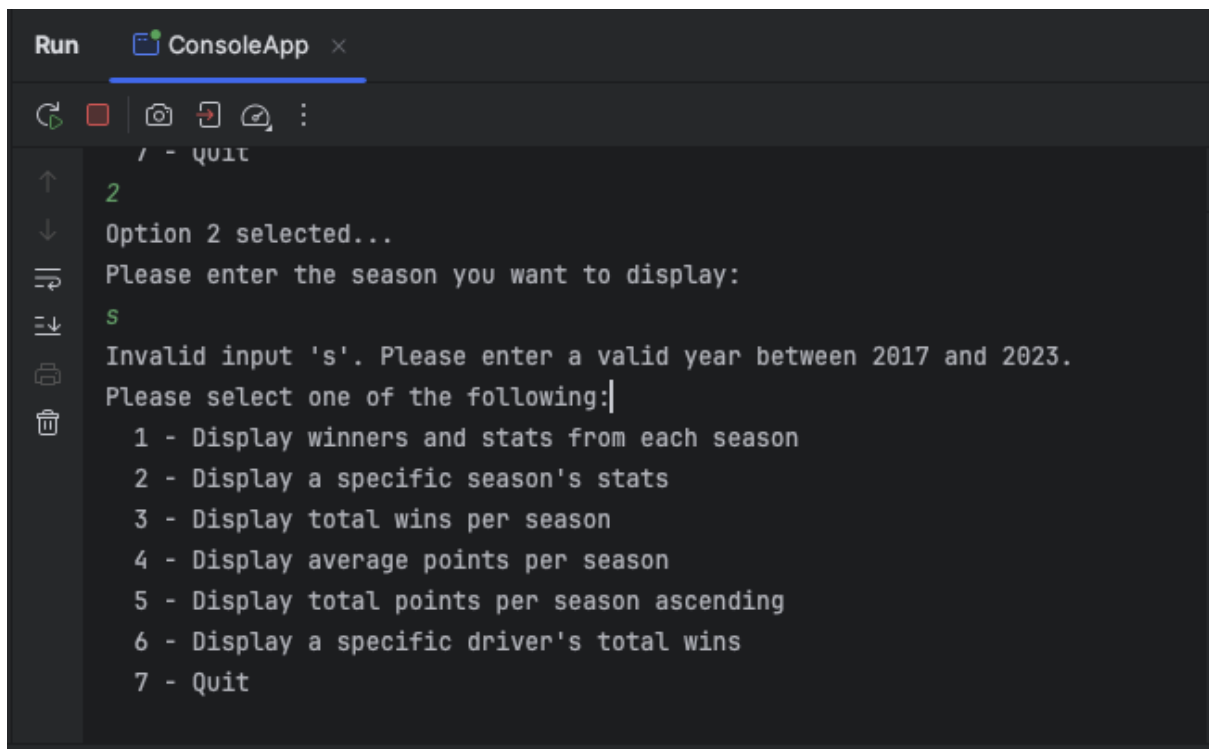
Figure 8

```
Run ConsoleApp x
Option 1 selected...
Season 2020: Winner: Lewis Hamilton, Points: 347.0, Wins: 11
Season 2017: Winner: Lewis Hamilton, Points: 363.0, Wins: 9
Season 2023: Winner: Max Verstappen, Points: 575.0, Wins: 19
Season 2022: Winner: Max Verstappen, Points: 454.0, Wins: 15
Season 2019: Winner: Lewis Hamilton, Points: 413.0, Wins: 11
Season 2021: Winner: Max Verstappen, Points: 395.5, Wins: 10
Season 2018: Winner: Lewis Hamilton, Points: 408.0, Wins: 11
Please select one of the following:
  1 - Display winners and stats from each season
  2 - Display a specific season's stats
  3 - Display total wins per season
  4 - Display average points per season
  5 - Display total points per season ascending
  6 - Display a specific driver's total wins
  7 - Quit
```

Figure 9

```
2
Option 2 selected...
Please enter the season you want to display:
2017
Season: 2017:
Driver: Lewis Hamilton, Points: 363.0, Wins: 9
Driver: Sebastian Vettel, Points: 317.0, Wins: 5
Driver: Valtteri Bottas, Points: 305.0, Wins: 3
Driver: Kimi Raikkonen, Points: 205.0, Wins: 0
Driver: Daniel Ricciardo, Points: 200.0, Wins: 1
Driver: Max Verstappen, Points: 168.0, Wins: 2
Driver: Sergio Perez, Points: 100.0, Wins: 0
Driver: Esteban Ocon, Points: 87.0, Wins: 0
Driver: Carlos Sainz, Points: 54.0, Wins: 0
Driver: Nico Hulkenberg, Points: 43.0, Wins: 0
Driver: Felipe Massa, Points: 43.0, Wins: 0
Driver: Lance Stroll, Points: 40.0, Wins: 0
Driver: Romain Grosjean, Points: 28.0, Wins: 0
Driver: Kevin Magnussen, Points: 19.0, Wins: 0
Driver: Fernando Alonso, Points: 17.0, Wins: 0
Driver: Stoffel Vandoorne, Points: 13.0, Wins: 0
Driver: Jolyon Palmer, Points: 8.0, Wins: 0
Driver: Pascal Wehrlein, Points: 5.0, Wins: 0
Driver: Daniil Kvyat, Points: 5.0, Wins: 0
Driver: Marcus Ericsson, Points: 0.0, Wins: 0
Driver: Pierre Gasly, Points: 0.0, Wins: 0
Driver: Antonio Giovinazzi, Points: 0.0, Wins: 0
Driver: Brendon Hartley, Points: 0.0, Wins: 0
Driver: Jenson Button, Points: 0.0, Wins: 0
Driver: Paul di Resta, Points: 0.0, Wins: 0
Please select one of the following:
  1 - Display winners and stats from each season
  2 - Display a specific season's stats
  3 - Display total wins per season
  4 - Display average points per season
  5 - Display total points per season ascending
  6 - Display a specific driver's total wins
  7 - Quit
```

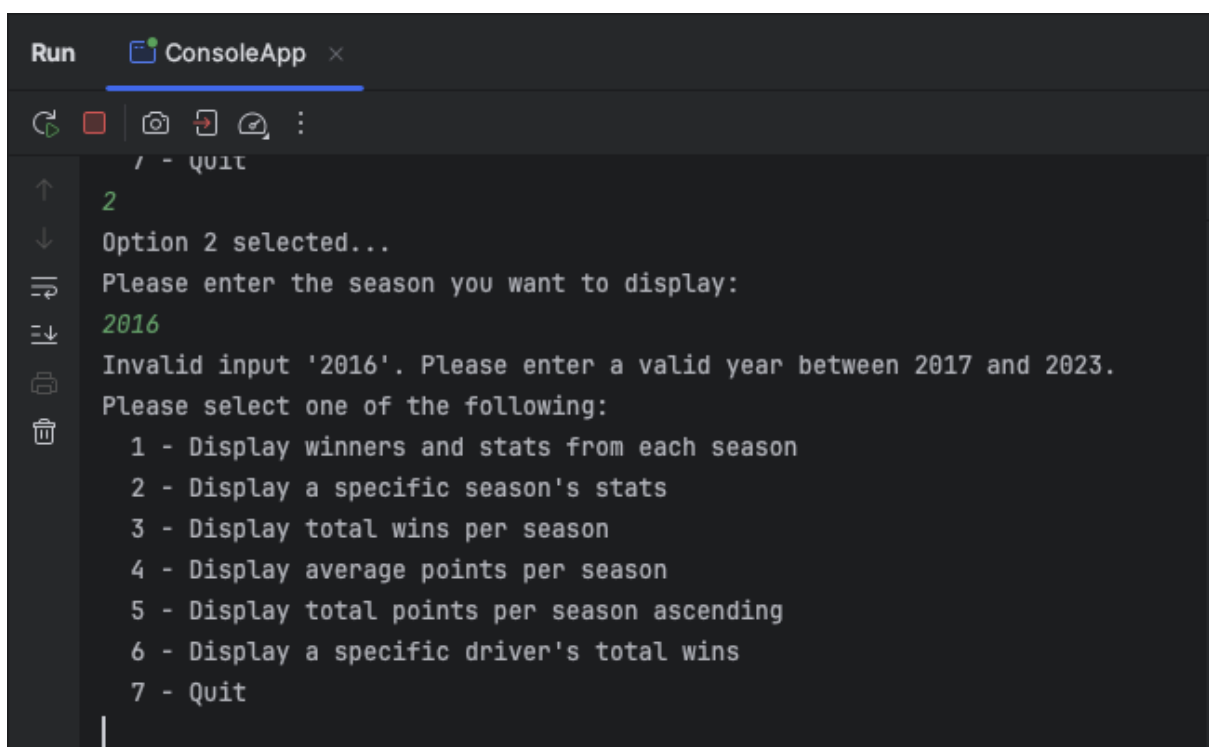
Figure 10



The screenshot shows a console application window titled "ConsoleApp". The application is running and displays a menu with seven options. The user has entered '2' to select an option, and then 's' to enter a season. The application responds with an error message indicating that 's' is not a valid year between 2017 and 2023.

```
Run ConsoleApp x
/ - quit
2
Option 2 selected...
Please enter the season you want to display:
s
Invalid input 's'. Please enter a valid year between 2017 and 2023.
Please select one of the following:
  1 - Display winners and stats from each season
  2 - Display a specific season's stats
  3 - Display total wins per season
  4 - Display average points per season
  5 - Display total points per season ascending
  6 - Display a specific driver's total wins
  7 - Quit
```

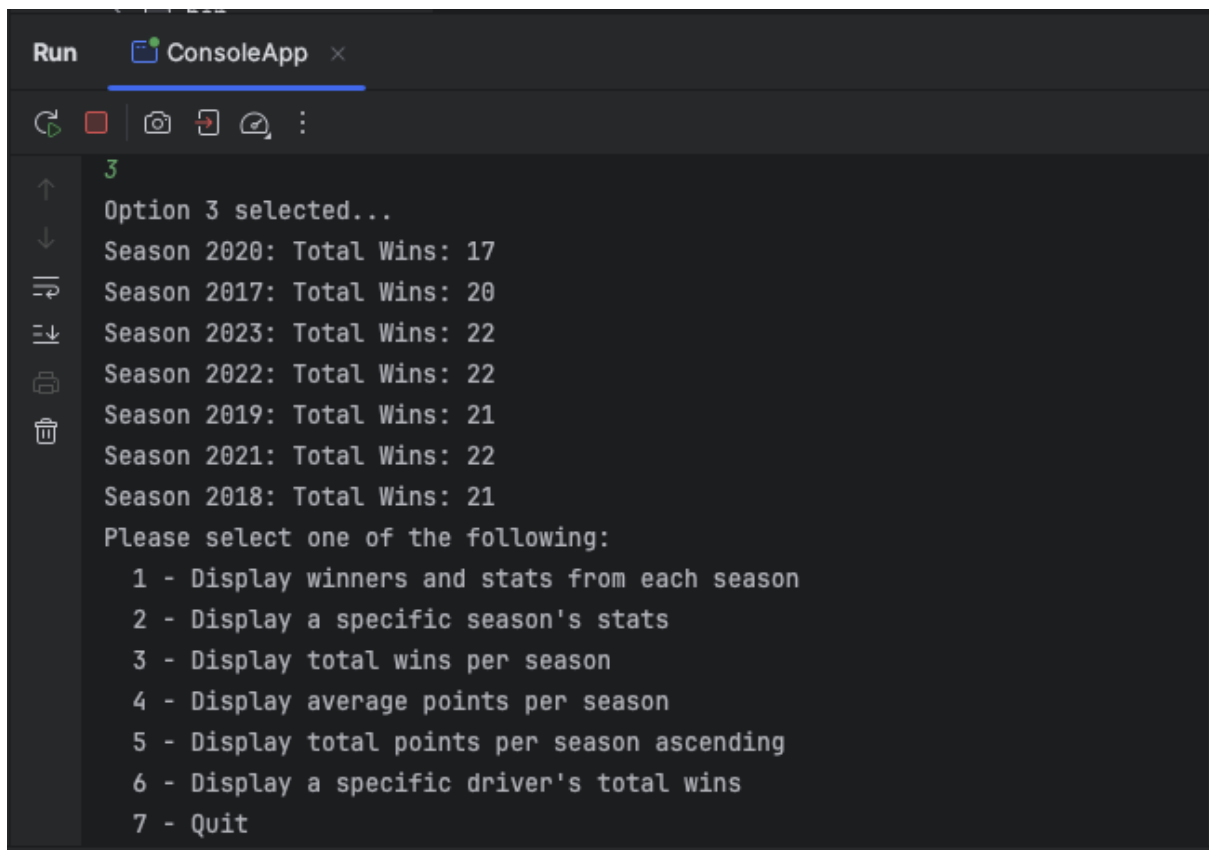
Figure 11



The screenshot shows the same console application window as Figure 11. The user has entered '2016' to enter a season. The application responds with an error message indicating that '2016' is not a valid year between 2017 and 2023. The menu is displayed again, and the user's cursor is at the bottom of the screen.

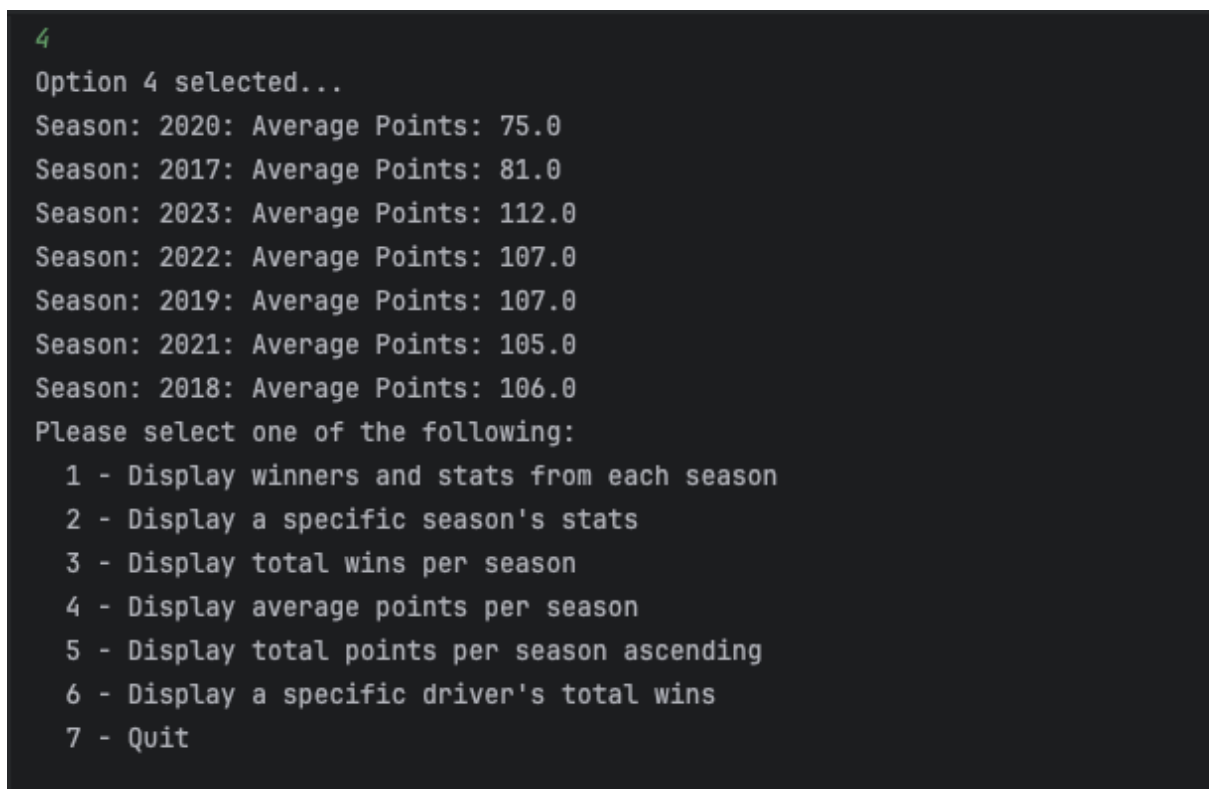
```
Run ConsoleApp x
/ - quit
2
Option 2 selected...
Please enter the season you want to display:
2016
Invalid input '2016'. Please enter a valid year between 2017 and 2023.
Please select one of the following:
  1 - Display winners and stats from each season
  2 - Display a specific season's stats
  3 - Display total wins per season
  4 - Display average points per season
  5 - Display total points per season ascending
  6 - Display a specific driver's total wins
  7 - Quit
|
```

Figure 12



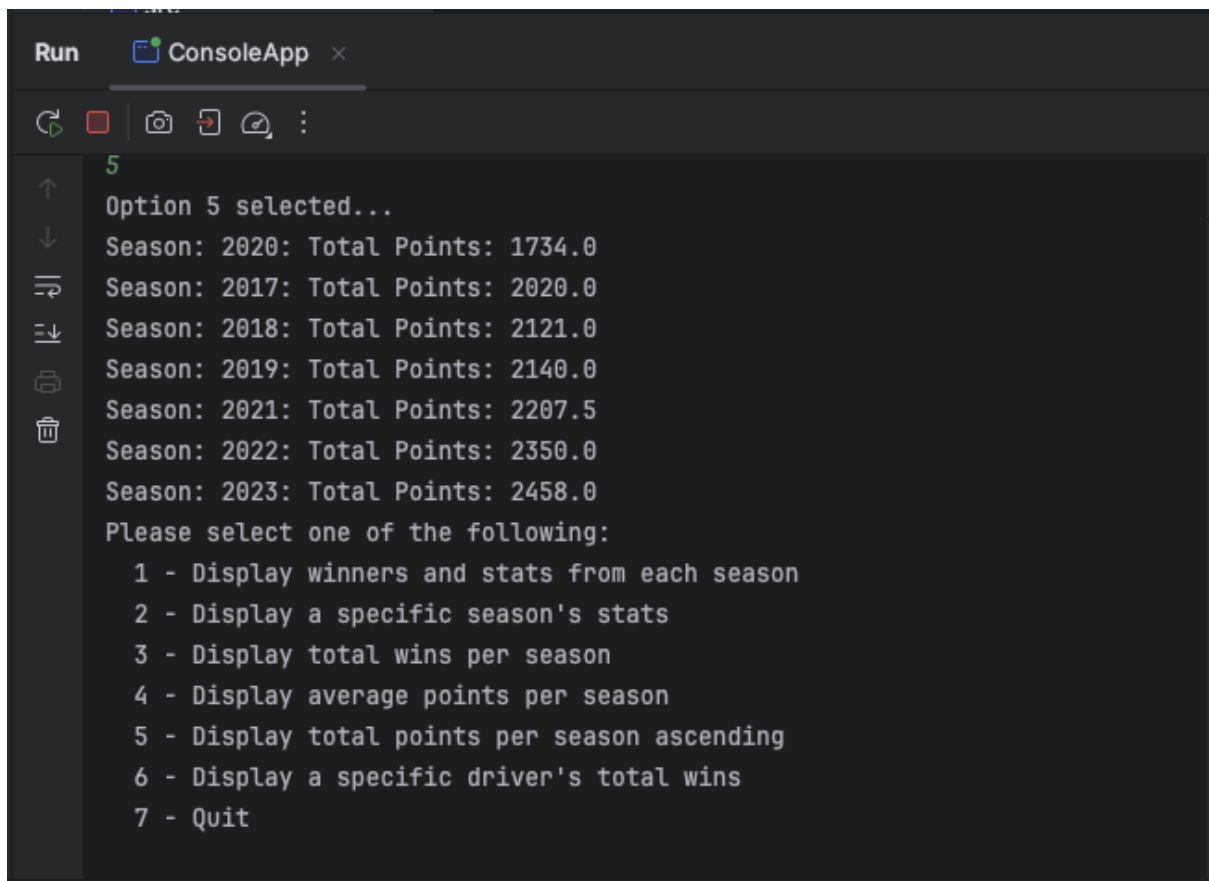
```
Run ConsoleApp x
Option 3 selected...
Season 2020: Total Wins: 17
Season 2017: Total Wins: 20
Season 2023: Total Wins: 22
Season 2022: Total Wins: 22
Season 2019: Total Wins: 21
Season 2021: Total Wins: 22
Season 2018: Total Wins: 21
Please select one of the following:
  1 - Display winners and stats from each season
  2 - Display a specific season's stats
  3 - Display total wins per season
  4 - Display average points per season
  5 - Display total points per season ascending
  6 - Display a specific driver's total wins
  7 - Quit
```

Figure 13



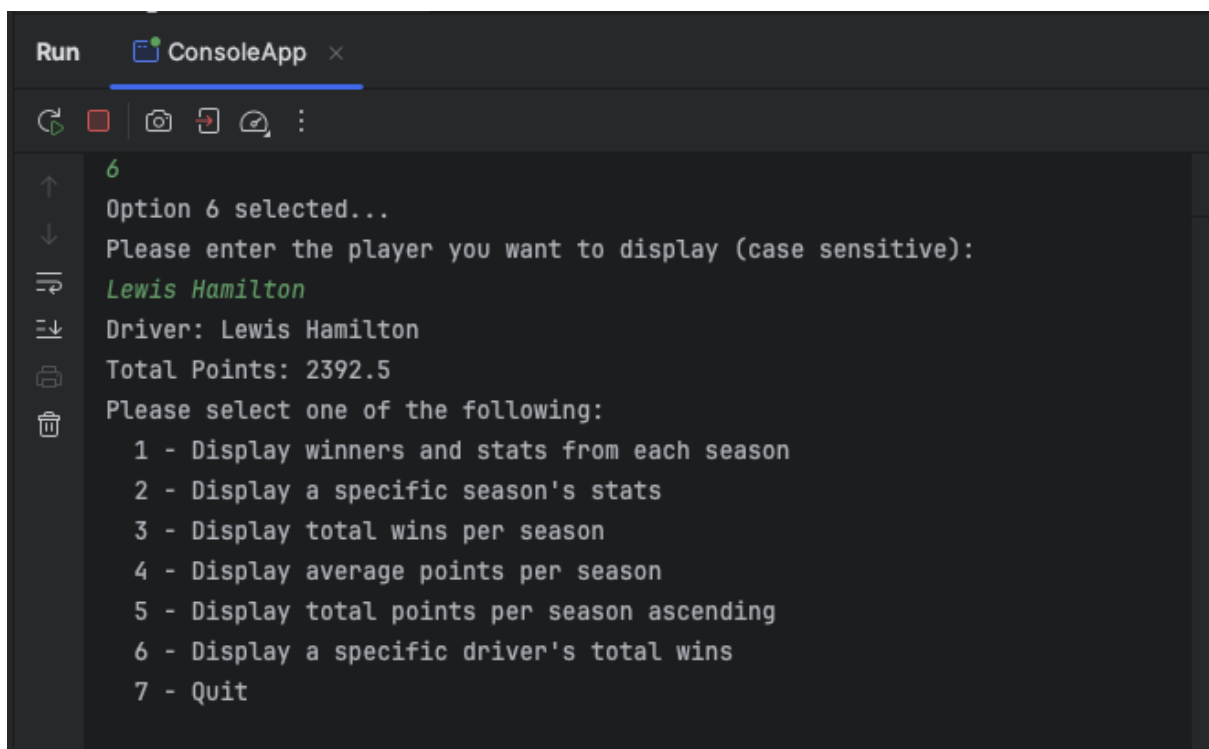
```
Option 4 selected...
Season: 2020: Average Points: 75.0
Season: 2017: Average Points: 81.0
Season: 2023: Average Points: 112.0
Season: 2022: Average Points: 107.0
Season: 2019: Average Points: 107.0
Season: 2021: Average Points: 105.0
Season: 2018: Average Points: 106.0
Please select one of the following:
  1 - Display winners and stats from each season
  2 - Display a specific season's stats
  3 - Display total wins per season
  4 - Display average points per season
  5 - Display total points per season ascending
  6 - Display a specific driver's total wins
  7 - Quit
```

Figure 14



```
Run ConsoleApp x
5
Option 5 selected...
Season: 2020: Total Points: 1734.0
Season: 2017: Total Points: 2020.0
Season: 2018: Total Points: 2121.0
Season: 2019: Total Points: 2140.0
Season: 2021: Total Points: 2207.5
Season: 2022: Total Points: 2350.0
Season: 2023: Total Points: 2458.0
Please select one of the following:
  1 - Display winners and stats from each season
  2 - Display a specific season's stats
  3 - Display total wins per season
  4 - Display average points per season
  5 - Display total points per season ascending
  6 - Display a specific driver's total wins
  7 - Quit
```

Figure 15



```
Run ConsoleApp x
6
Option 6 selected...
Please enter the player you want to display (case sensitive):
Lewis Hamilton
Driver: Lewis Hamilton
Total Points: 2392.5
Please select one of the following:
  1 - Display winners and stats from each season
  2 - Display a specific season's stats
  3 - Display total wins per season
  4 - Display average points per season
  5 - Display total points per season ascending
  6 - Display a specific driver's total wins
  7 - Quit
```

Figure 16

```
Run ConsoleApp x
6
Option 6 selected...
Please enter the player you want to display (case sensitive):
2
Invalid input '2'. The name wasn't found. Please enter a valid name from the list: Lewis Hamilton, Valtteri:
Please select one of the following:
1 - Display winners and stats from each season
2 - Display a specific season's stats
3 - Display total wins per season
4 - Display average points per season
5 - Display total points per season ascending
6 - Display a specific driver's total wins
7 - Quit
```

Figure 17

```
Run ConsoleApp x
Option 6 selected...
Please enter the player you want to display:
Lewis hamilton
Driver: Lewis Hamilton
Total Points: 2392.5
Please select one of the following:
1 - Display winners and stats from each season
2 - Display a specific season's stats
3 - Display total wins per season
4 - Display average points per season
5 - Display total points per season ascending
6 - Display a specific driver's total points
7 - Quit
```

Figure 18

```
6
Option 6 selected...
Please enter the player you want to display (case sensitive):

Invalid input ''. The name wasn't found. Please enter a valid name from the list: Lewis Hamilton, Valtteri
Please select one of the following:
1 - Display winners and stats from each season
2 - Display a specific season's stats
3 - Display total wins per season
4 - Display average points per season
5 - Display total points per season ascending
6 - Display a specific driver's total wins
7 - Quit
```

Figure 19



The screenshot shows a console application window titled "ConsoleApp". The output text is as follows:

```
option 0 selected...
Please enter the player you want to display:
Paul di Resta
Driver: Paul di Resta
Total Points: 0.0
Please select one of the following:
1 - Display winners and stats from each season
2 - Display a specific season's stats
3 - Display total wins per season
4 - Display average points per season
5 - Display total points per season ascending
6 - Display a specific driver's total points
7 - Quit
```

Figure 20

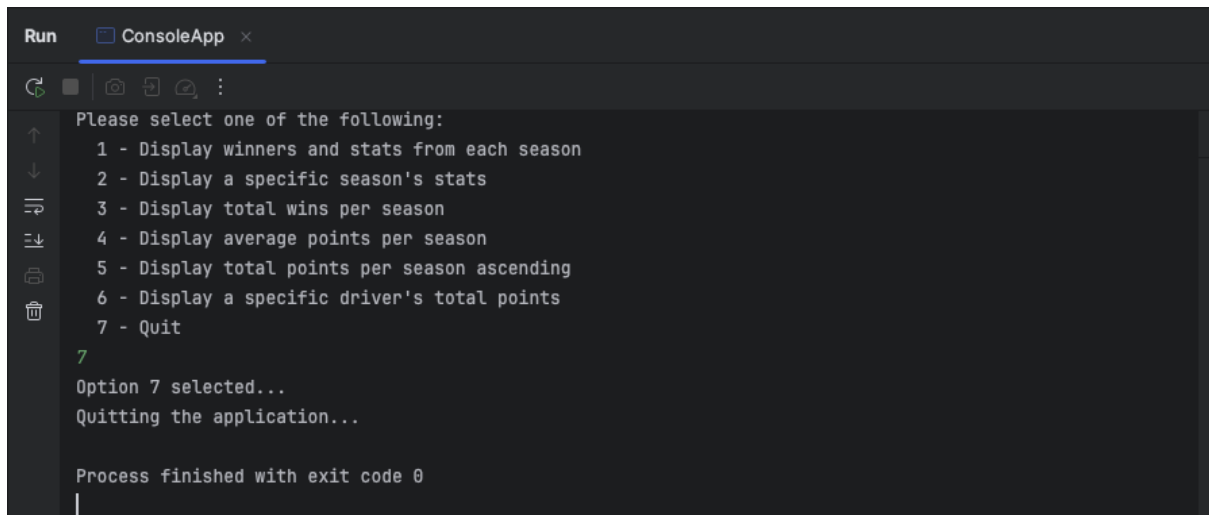
This screenshot is identical to Figure 20, showing the same console application output for Paul di Resta.

Figure 21

The screenshot shows the console application window with the following output:

```
option 0 selected...
Please enter the player you want to display:
Lewis Hamilton
Driver: Lewis Hamilton
Total Points: 2392.5
Please select one of the following:
1 - Display winners and stats from each season
2 - Display a specific season's stats
3 - Display total wins per season
4 - Display average points per season
5 - Display total points per season ascending
6 - Display a specific driver's total points
7 - Quit
```

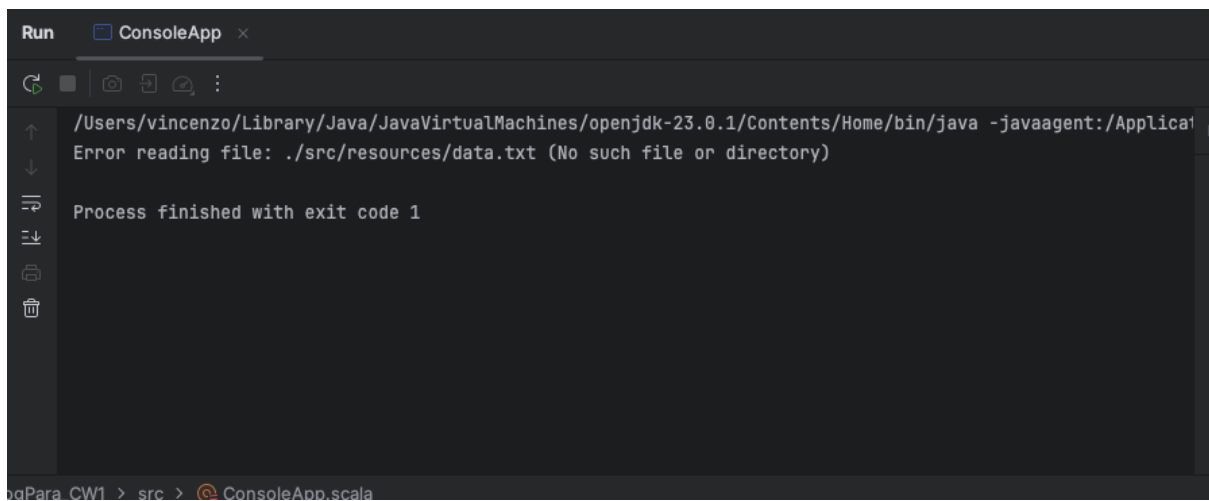
Figure 22



```
Run ConsoleApp x
Please select one of the following:
1 - Display winners and stats from each season
2 - Display a specific season's stats
3 - Display total wins per season
4 - Display average points per season
5 - Display total points per season ascending
6 - Display a specific driver's total points
7 - Quit
7
Option 7 selected...
Quitting the application...

Process finished with exit code 0
```

Figure 23



```
Run ConsoleApp x
/Users/vincenzo/Library/Java/JavaVirtualMachines/openjdk-23.0.1/Contents/Home/bin/java -javaagent:/Applica
Error reading file: ./src/resources/data.txt (No such file or directory)

Process finished with exit code 1
```

bdPara\_CW1 > src > ConsoleApp.scala

Figure 24



```
Run ConsoleApp x
/Users/vincenzo/Library/Java/JavaVirtualMachines/openjdk-23.0.1/Contents/Home/bin/java -javaagent:/Applica
Error: The data file is empty.

Process finished with exit code 1
```

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).