Exercise 5

Type Parameterization in Scala

- Last Week Recap [Collections(*A collection is a single object representing a group of objects and they should be used when we want to associate some significant **behaviour** with the data in the collection*) i.e. **Sequence, Set and Map**]
- **Pattern Matching** (*It allows you to match values against patterns and execute code based on the match. Pattern matching is often used with case classes and case objects, which are commonly used in Scala for modeling data.*)
- **Type Parameterization** (*It allows creating generic code that can work with different types of data. This is an essential concept in functional programming as it enables the creation of reusable, type-safe, and concise code.*)
- **Variance** (*A mechanism for defining subtyping relationships between generic types. It determines whether a type parameter can be replaced by a subtype or a supertype. This is important in functional programming because it helps to ensure type safety and prevent runtime errors.*)
  Scala defines three variance annotations that can be applied to type parameters: **+**, **-**, and **=**.
  +: Covariant annotation: It means that a type parameter can be replaced by a ***subtype***. For example, **List[+A]** is covariant in A, which means that a **List[Int]** can be treated as a **List[AnyVal]**. This is useful when defining immutable collections.

  -: Contravariant annotation: It means that a type parameter can be replaced by a ***supertype***. For example, **Function1[-A, +B]** is contravariant in A and covariant in B. This means that a function of type **AnyVal => String** can be treated as a function of type **Int => AnyRef**. This is useful when defining functions that consume values of a certain type.

  =: Invariant annotation: It means that a type parameter cannot be replaced by a ***subtype*** or a ***supertype*** of its bound. For example, **Option[A]** is invariant in A, which means that an **Option[Int]** cannot be treated as an **Option[AnyVal]** or vice versa. This is useful when defining types that should have strict type checking.

Classwork

In this lab, you will be exploring the fundamentals of type parameterisation, variance and pattern matching in Scala. You will also learn about the type constraints and practice how to use them in Higher Order Functions (HOFs) in a FP manner.

Homework

NOTE: You should use immutable data types and avoid using any mutable variables or loops.

Task 1: Type Parameterisation

Implement a generic function that can compute the average of a list of values of any numeric

type in a recursive manner. The function should take a list of type **A** and return a value of type **A**, where **A** is any numeric type such as **int**, **float**, or **double**. The function should be tested with different input lists and the output should be verified to ensure that the average is computed correctly for each list.

Task 2: Variance

Define a class hierarchy of animals, with a base class **Animal** and two derived classes **Bird** and **Mammal**. Add a method ***makeSound*** to each class that returns a string representing the sound the animal makes. Implement a function ***makeAllSounds*** that takes a list of animals and returns a list of strings representing the sounds each animal makes. Use covariance to ensure that the return type of the ***makeAllSounds*** function is a list of strings, even though the input list may contain objects of different classes. Test the function with a list of birds and a list of mammals to ensure that the sounds are correctly returned for each animal.

Task 3: Type Constraints

Create a **typeclass** that represents a collection of objects that can be sorted. The **typeclass** should have a single recursive function ***sort*** that takes a list of objects and returns a sorted list. Implement instances of the **typeclass** for several types, including **integers, floating-point numbers, and strings**. Use type constraints to ensure that the objects in the list have a total ordering, which is necessary for sorting. Test the sort function with input lists of different types and sizes to ensure that it correctly sorts the lists according to the total ordering defined for each type.

Task 4: Pattern Matching

Write a function **isPalindrome** that takes a string as input and returns true if the string is a palindrome (i.e., it reads the same backward as forward), and false otherwise. Use pattern matching to implement the function.

[Hint: You can use pattern matching to match the first and last characters of the string, and recursively match the substring between them until the string is empty or has one character left.]

Examples

isPalindrome("rotator") // true

isPalindrome("scala") // false

Deliverable

Deliverable: Submit a single scala code file with "*.scala*" extension, and write your Name and Student ID in the code as a comment. The whole deliverable must be well commented and supported with descriptions where required.

Deadline: 21.04.2023 12:00 am [Before Next Friday Session]

Submission: (session respective) Return box on Moodle.

Estimated workload: <= 2 hours

Warning: This is individual work. Strict actions will be taken for plagiarism!

Deliverables for Exercise 5:

1. Implementation of the homework part.

Note:

Make sure to follow the Scala style guide and use appropriate naming conventions for variables, functions, and classes. Your code should be well-organized, well-documented, and easy to read.