Kinza Fatima

LEARNING JOURNEY

FUNCTIONAL PROGRAMMING IN SCALA

Topis:

- Scala Traits and Inheritance
- Linear flattening
- Modules as parameters in SCALA
- Implicit Function, Class and Parameters
- Lazy Val

1. **Scala Traits and Inheritance**:
    - **Traits** are a fundamental feature in Scala that allow you to define methods and fields that can be shared across multiple classes. They're similar to interfaces in other languages but can also contain concrete implementations.

      Example:

```
trait Animal {
  def makeSound(): Unit
}
trait CanFly {
  def fly(): Unit = {
println("Flying!")
}
}
class Bird extends Animal with CanFly {
  def makeSound(): Unit = println("Chirp chirp")
}
val bird = new Bird
bird.makeSound()  // Chirp chirp
bird.fly()        // Flying!
```

- **Inheritance** in Scala is straightforward, allowing classes to extend other classes. Scala supports single inheritance for classes and multiple inheritance for traits.

  Example:

```
class Animal {
  def makeSound(): Unit = println("Some sound")
}

class Dog extends Animal {
  override def makeSound(): Unit = println("Bark bark")
}

val dog = new Dog
dog.makeSound()  // Bark bark
```

2. **Linear Flattening**:
   - This refers to the process of transforming hierarchical or nested structures into a flat, linear structure. In the context of Scala, it might involve flattening nested lists or other collections to simplify data processing.

     Example:

Flattening a Nested List:

```
val nestedList = List(List(1, 2), List(3, 4), List(5))
val flatList = nestedList.flatten
println(flatList)          // List(1, 2, 3, 4, 5)
```

Flattening Nested Options:

```
val nestedOptions = List(Some(1), None, Some(2))
val flatOptions = nestedOptions.flatten
println(flatOptions)       // List(1, 2)
```

//Removes `None` values and extracts values from `Some`

3. **Modules as Parameters in Scala**:
    - In Scala, you can pass modules (or classes/objects) as parameters to functions or constructors. This allows for flexible and reusable code, enabling you to abstract and encapsulate behavior more effectively.

      Example:

```
object MathUtils {
  def add(x: Int, y: Int): Int = x + y
}

class Calculator(val mathUtils: MathUtils.type) {
  def calculateSum(x: Int, y: Int): Int = mathUtils.add(x, y)
}

val calculator = new Calculator(MathUtils)
println(calculator.calculateSum(5, 3))                    // 8
```

4. **Implicit Function, Class, and Parameters**:
    - **Implicit Parameters**: These are parameters that are automatically passed to functions or constructors if they're not explicitly provided. They're used to make code more concise and readable.

```
def greet(name: String)(implicit greeting: String): Unit = {
  println(s"$greeting, $name!")
}

implicit val defaultGreeting: String = "Hello"

greet("Alice")  // Hello, Alice!
```

    - **Implicit Classes**: These are classes that are used to add methods to existing types without modifying them directly. They can enhance the usability of existing libraries or APIs.

```
implicit class RichString(val str: String) {
  def shout(): String = str.toUpperCase + "!!!"
}

println("hello".shout())  // HELLO!!!
```

- **Implicit Functions**: These are functions that are used by the compiler to automatically convert one type to another. They can be useful for type conversions and ensuring code flexibility.

```
implicit def intToString(x: Int): String = x.toString

val num: String = 42
println(num)  // 42
```

5. **Lazy Val**:

Lazy Values are variables that are initialized only when they are first accessed. This can help improve performance by delaying computation until it's actually needed. This is particularly useful for expensive computations or to avoid initializing values unnecessarily.

```
lazy val expensiveComputation: Int = {
  println("Performing expensive computation...")
  42              // Simulate a time-consuming computation
}

println("Before accessing lazy val")
println(expensiveComputation)                  // Performing expensive computation... 42
println(expensiveComputation)                   // 42 (computed only once)
```

First Access:

- When `expensiveComputation` is accessed for the first time, the initialization block is executed.
- The message `"Performing expensive computation..."` is printed, indicating that the computation is taking place.
- The result of the computation (`42`) is returned and printed.

Subsequent Accesses:

- On subsequent accesses to `expensiveComputation`, the initialization block is **not** executed again.
- Instead, the previously computed value (`42`) is directly returned.
- This avoids re-running the expensive computation and only shows `42` without the initialization message.