LEARNING JOURNEY

FUNCTIONAL PROGRAMMING IN SCALA

Scala Basics

Concepts:

- **Immutable Data**: Variables in Scala can be declared as `val` (immutable) or `var` (mutable).
- **Functions as Values**: Functions are first-class citizens, meaning they can be assigned to variables, passed as arguments, and returned from other functions.
- **Higher-Order Functions**: Functions that take other functions as parameters or return functions as results.
- **Pattern Matching**: A powerful feature for deconstructing data structures.

Syntax:
Immutable Variables:

```
val x = 10   // Immutable variable
// x = 20   // Error: Reassignment to val
```

Functions:

```
def add(a: Int, b: Int): Int = {
a + b
}
```

Higher-Order Functions:

```
def operateOnNumbers(x: Int, y: Int, operation: (Int, Int) => Int):
Int = {
  operation(x, y)
}
// Usage:
val result = operateOnNumbers(3, 5, (a, b) => a * b)
```

Pattern Matching:

```
def matchExample(x: Any): String = x match {
  case 1 => "One"
  case "hello" => "Greeting"
```

```
  case _: Int => "An integer"
  case _ => "Something else"
}
```

Dive into Functional Programming in Scala

Concepts:

- **Immutable Collections**: Lists, Sets, Maps, etc., which are immutable by default.
- **Functional Constructs**: `map`, `filter`, `reduce`, `flatMap`, `for-comprehensions`.

Syntax:
Immutable Collections:

```
val list = List(1, 2, 3, 4, 5)

val myArray = Vec(size, elementType)
val myArray = Vec(4, UInt(8.W)) // Array of 4, 8-bit unsigned integers

val mappedList = list.map(x => x * 2)
 .map: This is the map function applied to the list.
- (x => x * 2): This defines a function that takes one argument, x,
and returns its value multiplied by 2, it doubles the value of x.
```

```
val originalList = List(1, 2, 3, 4)
val mappedList = originalList.map(x => x * 2)
println(mappedList)            // This will print List(2, 4, 6, 8)
```

```
val filteredList = list.filter(_ % 2 == 0)
```

filter(_ % 2 == 0): This part applies the filter function to the list.

- _ % 2 == 0: This is a shorthand way of writing a function that takes one argument (represented by _) and checks if it's divisible by 2 (even). If it's even, the element is included in the new list; otherwise, it's excluded.

Functional Constructs:

```
val sum = list.reduce((x, y) => x + y)
val flattenedList = list.flatMap(x => List(x, x * 2))
val evenSquares = for {
  x <- list
  if x % 2 == 0
} yield x * x
```

val sum = list.reduce((x, y) => x + y)

- **reduce**: This function applies a binary operation to an entire list, reducing it to a single value.
- **(x, y) => x + y**: This is a lambda function that takes two elements, x and y, and returns their sum.
- **sum**: This variable will hold the final sum of all elements in the list.

```
val numbers = List(1, 2, 3, 4)

 val sum = numbers.reduce((x, y) => x + y)              // sum will be 10
```

Understand Chisel Basics

Concepts:

- **Hardware Description Languages**: Concepts of designing digital circuits using code.
- **Chisel Syntax**: Modules, Wires, Registers, etc., for describing hardware components.

Syntax:
Chisel Module:

```
import chisel3._

class MyModule extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(4.W))
    val out = Output(UInt(4.W))
```

```
  })

  val reg = RegInit(0.U)
  reg := io.in

  io.out := reg
}
```

Combine Scala and Chisel

Concepts:

- **Functional Hardware Design**: Applying functional programming principles within Chisel modules.
- **Immutable State**: Designing hardware components without mutable state.

Example:
Combining Functional Programming and Chisel:
scala
Copy code

```
import chisel3._

class MyAdder extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(4.W))
    val b = Input(UInt(4.W))
    val result = Output(UInt(4.W))
  })

  def add(a: UInt, b: UInt): UInt = {
    a + b
  }

  io.result := add(io.a, io.b)
}
```

EXERCISE 1:

Write a Scala function named `fib` that recursively calculates the nth Fibonacci number using tail recursion. The first two Fibonacci numbers are 0 and 1. The function should have the following signature: def fib(n: Int): Int

```
import scala.::
import scala.util.Random

object Exercise1{
   def fib(n: Int): Int = {
       if (n <= 1)
       n
       else {
       var a = 0
       var b = 1

       var c = 0
       for (i <- 2 to n) {
           c = a + b
           a = b
           b = c
       }
       c
 }
}
   def main(args: Array[String]): Unit = {
       println(fib(9))
   }
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                    java

[success] Total time: 1 s, completed Jul 25, 2024, 7:13:07 PM
sbt:Scala-Chisel-Learning-Journey> run
[info] compiling 3 Scala sources to /home/kinzaa/Functional-Programming/target/scala-2.12/classes ...
[info] running Exercise1
34
[success] Total time: 2 s, completed Jul 25, 2024, 7:21:00 PM
sbt:Scala-Chisel-Learning-Journey>
```

Higher-Order Functions

A higher-order function is a function that does at least one of the following:

- Takes one or more functions as arguments
- Returns a function as its result

```
val numbers = List(1, 2, 3, 4, 5)
// Using map
val doubledNumbers = numbers.map(_ * 2)
// Using filter
val evenNumbers = numbers.filter(_ % 2 == 0)
// Using reduce
val sum = numbers.reduce(_ + _)
```

Example:

```
val numbers = List(1, 2, 3, 4, 5)

val doubledNumbers = numbers.map(x => x * 2)
println(doubledNumbers)              // Output: List(2, 4, 6, 8, 10)
```

Monomorphic & Polymorphic functions

- ## Monomorphic Functions

  A monomorphic function is a function that operates on a single, specific data type. It is specialized to work with that particular type and cannot be applied to other data types.

  ```
  def add(x: Int, y: Int): Int = x + y
  ```
  Example:
  ```
  def isEven(n: Int): Boolean = {
          n % 2 == 0
  }
  ```

- ## Polymorphic Functions

  A polymorphic function is a function that can work with multiple data types. It is generic and can be applied to different types, making it more flexible and reusable.

  ```
  def identity[A](x: A): A = x
  ```

  Example:

  ```
  def first[A](list: List[A]): Option[A] = list match {

    case Nil => None

    case x :: _ => Some(x)

  }
  ```

Feature	Monomorphic Functions	Polymorphic Functions
Data type	Single, specific type	Multiple data types
Flexibility	Less flexible	More flexible
Reusability	Less reusable	More reusable
Type safety	Type-safe	Type-safe

Understanding Currying and Uncurrying

- **Currying** is a technique of transforming a function that takes multiple arguments into a nested function that takes one argument at a time.

  ```
  def add(x: Int)(y: Int): Int = x + y

  // Curried function:

   val curriedAdd: Int => (Int => Int) = add
  ```

- **Uncurrying** is the reverse process, taking a curried function and transforming it back into a function that takes multiple arguments.

  ```
  def uncurry[A, B, C](f: A => B => C): (A, B) => C = (a, b) => f(a)(b)

  val uncurriedAdd: (Int, Int) => Int = uncurry(add)
  ```

Scala Collections: A Powerful Toolset

Scala collections provide a comprehensive framework for handling and manipulating data structures. They are designed to be efficient, flexible, and type-safe.

Mutable vs. Immutable Collections

1. Immutable Collections (default collections)

- **Unchangeable:** Once created, an immutable collection cannot be modified. Any operation that appears to modify it actually creates a new collection.
- **Thread-safe:** Because they cannot be changed, they are inherently thread-safe.
- **Functional programming paradigm:** Align well with functional programming principles, promoting immutability and pure functions.
- **Common operations:** `map`, `filter`, `flatMap`, `reduce`, etc.
- **Examples:** `List`, `Vector`, `Set`, `Map`

When to use:

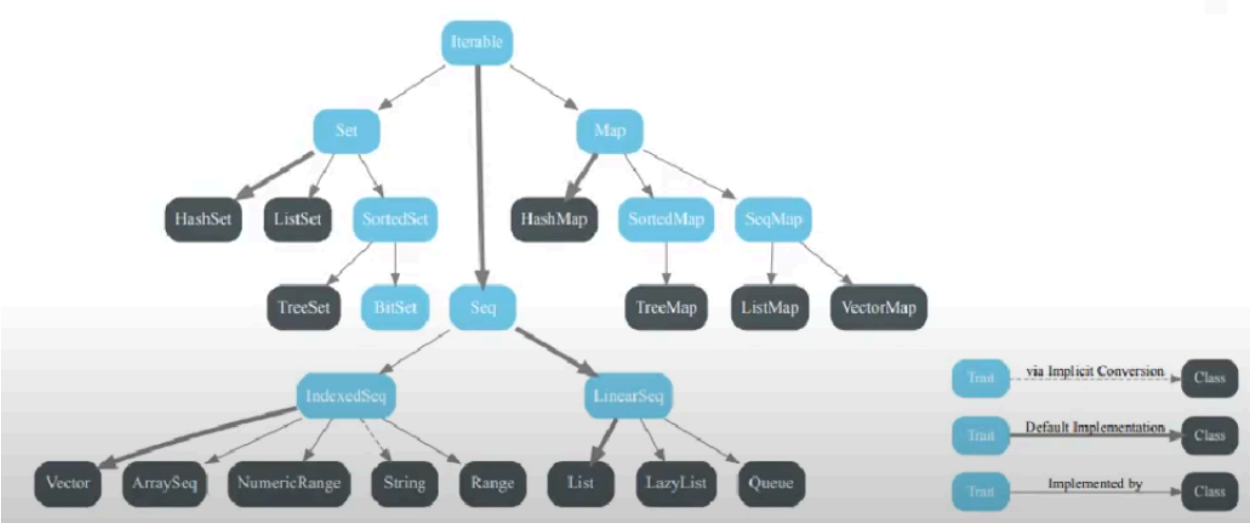
- When you need thread safety.
- When you want to avoid side effects and promote functional programming.
- When you need to preserve the original collection for later use.

2. Mutable Collections

- **Changeable:** Elements can be added, removed, or modified in place.
- **Not thread-safe:** Requires explicit synchronization for concurrent access.
- **Imperative programming style:** Often used in imperative programming constructs.
- **Common operations:** `+=`, `-=`, `update`, etc.
- **Examples:** `ArrayBuffer`, `ListBuffer`, `HashMap`, `HashSet`

When to use:

- When performance is critical and you need to modify a collection in place.
- When you are building data structures that need to be efficiently updated.
- When you are comfortable with managing mutable state and potential thread safety issues.



Self study topics

Array, list, tuple, set

Arrays

- **Fixed-size collections** of elements of the same type.
- Accessed by index, starting from 0.
- Mutable by default.
- Efficient for random access but less efficient for insertions or deletions.

```
val numbers = Array(1, 2, 3, 4)
numbers(2) = 5                  // Modify element at index 2
```

Tuples

- **Fixed-length collections** of elements of potentially different types.
- Immutable.
- Primarily used for returning multiple values from a function or method.

```
val person = ("Alice", 30, true)
println(person._1)              // Access first element
```

Sets

- **Unordered collections** of unique elements.
- Can be mutable or immutable.
- Efficient for checking membership and removing duplicates.

```
val immutableSet = Set(1, 2, 3, 2)              // Duplicate 2 is removed
val mutableSet = scala.collection.mutable.Set(1, 2, 3)
mutableSet += 4
```

Maps

A Map in Scala is a collection of key-value pairs. It's similar to a dictionary in other languages. Each key in a map must be unique, but values can be duplicated.

Types of Maps

There are two primary types of maps in Scala:

1. **Immutable Maps:** These maps cannot be modified once created. They are the default.
2. **Mutable Maps:** These maps can be modified after creation.

```
// Immutable Map

val colors = Map("red" -> "#FF0000", "green" -> "#00FF00", "blue" -> "#0000FF")

 // Mutable Map

import scala.collection.mutable.Map

val mutableColors = Map[String, String]() mutableColors += ("yellow" -> "#FFFF00")
```

Common Operations

- **`keys`**: Returns an iterable of all keys.
- **`values`**: Returns an iterable of all values.
- **`isEmpty`**: Checks if the map is empty.
- **`contains`**: Checks if a key exists in the map.
- **`map`**: Applies a function to each key-value pair.
- **`filter`**: Filters key-value pairs based on a predicate.

Iterators (similar to queue)

An iterator is a way to access the elements of a collection one by one. It's like a cursor that points to the current element and allows you to move forward to the next.

Basic Operations

- **`hasNext`**: Checks if there's another element to be returned.
- **`next`**: Returns the next element and advances the iterator.

```
val numbers = List(1, 2, 3, 4)
val iterator = numbers.iterator
while (iterator.hasNext){
        val element = iterator.next()
        println(element)
}
```

When to Use Iterators

- When you need to process elements one by one and potentially stop early based on a condition.
- When you need to create custom iterators for specific data structures.
- When you're working with infinite collections (e.g., streams).

Wildcards in Scala

In Scala, a wildcard, represented by an underscore (`_`), is a versatile construct used in various contexts. Let's explore its common applications:

What is the `apply` Method?

In Scala, the `apply` method is a special method that allows you to call an object as if it were a function. This means you can use parentheses after an object name, passing arguments to the `apply` method.

```
object Doubler {
  def apply(x: Int): Int = x * 2
}

val result = Doubler(5)                    // Equivalent to Doubler.apply(5)
println(result)                            // Output: 10
```

Zip, Unzip, and ZipWithIndex in Scala

Zip (zip method return list)

The `zip` method combines two collections element-wise into a new collection of pairs. If the collections have different lengths, the resulting collection will have the length of the shorter one.

```
val numbers = List(1, 2, 3)
val letters = List('a', 'b', 'c')

val pairs = numbers zip letters
// pairs: List[(Int, Char)] = List((1,a), (2,b), (3,c))
```

Unzip

The `unzip` method is the inverse of `zip`. It takes a collection of pairs and splits it into two collections.

```
val pairs: List[(Int, String)] = List((1, "one"), (2, "two"), (3, "three"))

val (numbers, strings) = pairs.unzip
// numbers: List[Int] = List(1, 2, 3)
// strings: List[String] = List(one, two, three)
```

ZipWithIndex

The `zipWithIndex` method pairs each element of a collection with its index.

```
val numbers = List(1, 2, 3)

val indexedNumbers = numbers.zipWithIndex
// indexedNumbers: List[(Int, Int)] = List((1,0), (2,1), (3,2))
```

What is `reduce`?

In Scala, `reduce` is a higher-order function that takes a sequence and a binary operation, and combines all elements of the sequence using the given operation into a single value.

```
val numbers = List(1, 2, 3, 4, 5)
val sum = numbers.reduce((x, y) => x + y)
println(sum)                    // Output: 15
val product = numbers.reduce((x, y) => x * y)
println(product)                // Output: 120
```

ReduceLeft

- Processes the elements from left to right.
- Takes an initial value as the accumulator.
- Typically more efficient for lists due to their structure.

```
val numbers = List(1, 2, 3, 4, 5)
val sumLeft = numbers.reduceLeft(0)(_ + _)          // 15
```

ReduceRight

- Processes the elements from right to left.
- Takes an initial value as the accumulator.
- Generally less efficient than reduceLeft for lists.

```
val numbers = List(1, 2, 3, 4, 5)
val sumRight = numbers.reduceRight(0)(_ + _)       // 15
```

Maps

A map in Scala is a collection of key-value pairs. Each key must be unique within the map, while values can be duplicated. Maps are essential for storing and retrieving data based on specific identifiers.

- **Immutable by default:** Scala's maps are immutable unless explicitly specified.
- **Key-value pairs:** Each element in a map consists of a key and its corresponding value.
- **Efficient lookup:** Maps provide efficient access to values based on their keys.
- **Multiple implementations:** Scala offers various map implementations like `Map`, `HashMap`, `TreeMap`, etc., with different performance characteristics.

Creating Maps

```
// Immutable Map
val colors = Map("red" -> 1, "green" -> 2, "blue" -> 3)

// Mutable Map (requires import)
import scala.collection.mutable.Map
val mutableColors = Map("red" -> 1, "green" -> 2, "blue" -> 3)
mutableColors += ("yellow" -> 4) // Adding a new element
```

FlatMap

FlatMap is a higher-order function in Scala that combines the `map` and `flatten` operations into a single step. It's particularly useful for transforming collections of collections into a single flat collection.

```
val list = List(List(1, 2), List(3, 4))
val flattenedList = list.flatMap(x => x)
println(flattenedList)              // Output: List(1, 2, 3, 4)
```