Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Software Engineering and Automation

Real Time Programming

PROJECT 0

Performed by:
Anna Chiriciuc
std. gr. FAF-201

Verified by:
Alex Osadchenko
assist. univ.

Chișinău 2023

1 VCS

GitHub repository: *https://github.com/AnnaWeber07/PTR/*

2 P0W1 - Welcome...

Minimal Task: Follow an installation guide to install the language / development environment of your choice.

I used IntellijIdea Ultimate with Akka support for Scala, JetBrains. Student license.

Minimal Task: Write a script that would print the message “Hello PTR” on the screen. Execute it.

```
  def Output(str: String): String = {
    println(str)
    str
  }
```

Listing 1: Hello PTR

Just print out the received string.

Bonus Task: Write a comprehensive readme for your repository.

Available on *https://github.com/AnnaWeber07/PTR*

Bonus Task: Create a unit test for your project. Execute it.

```
    val str = "Hello PTR"
    var checker = RiceFields.Output(str)
    println("Check the Hello PTR function: " + RiceFields.Verify(checker))
  }
```

Listing 2: Unit test

Check the output with the result.

3 P0W2 - ..to the rice fields

Minimal Task: Write a function that determines whether an input integer is prime.

isPrime(13) -> True

```
      def isPrime(primes: Int): Boolean = {
    //require(primes (_ >= 0), "negative number")
    if
    (primes <= 1)
      false
    else if (primes == 2)
      true
    else
      !(2 until primes).exists(n => primes % n == 0)
  }

     val x = 13
    println("is " + x + " prime?" + " " + RiceFields.isPrime(x))
    println()
```

Listing 3: isPrime

Check if the number is Prime.

Minimal Task: Write a function to calculate the area of a cylinder, given it's height and radius.

cylinderArea $(3, 4) \rightarrow 175.9292$

```
  def cylinderArea(height: Int, rad: Int): Double = {
    2 * Math.PI * rad * (rad + height)
  }
    val height = 3
    val radius = 4

    println("area of cylinder with height $height and radius $radius: ")
    println(RiceFields.cylinderArea(height, radius))
```

Listing 4: Cylinder Area

Calculate the cylinder area.

Minimal Task: Write a function to reverse a list.

reverse $([1, 2, 4, 8, 4]) \rightarrow [4, 8, 4, 2, 1]$

```
      def reversal(list: List[Int]): List[Int] = {
    list.reverse
  }

     val integers: List[Int] = List(1, 2, 4, 8, 4)

    println("Original list: " + integers)
    println("Reversed list: " + RiceFields.reversal(integers))
```

Listing 5: Reverse List

Reverse the list with built-in function.

Minimal Task: Write a function to calculate the sum of unique elements in a list.

1 uniqueSum $([1, 2, 4, 8, 4, 2]) \rightarrow 15$

```
      def uniqueSum(list: List[Int]): Int = {
    list.distinct.sum
  }
    val uniqueElements: List[Int] = List(1, 2, 4, 8, 4, 2)

    println("Elements in a list " + uniqueElements)
    println("Sum: " + RiceFields.uniqueSum(uniqueElements))
```

Listing 6: Unique Sum

Select distinct items and sum them.

Minimal Task: Write a function that extracts a given number of randomly selected elements from a list.

1 extractRandom $([1, 2, 4, 8, 4], 3) \rightarrow [8, 4, 4]$

```
        val random = new Random()
    var randomElements: List[Int] = List(1, 2, 4, 8, 4)
    val randomQuantity = random.nextInt(randomElements.size)

    println("Random elements quantity: " + randomQuantity)
    for (x <- 1 to randomQuantity) {
      print(random.nextInt(randomElements.length) + " ")
```

```
    }
```

Listing 7: Extract Random

Select random elements and their random quantity. Print them.

Minimal Task: Write a function that returns the first n elements of the Fibonacci sequence.

firstFibonaccielements $(7) \rightarrow [1, 1, 2, 3, 5, 8, 11]$

```

 def fibonacciNumbers(a: Int = 0, b: Int = 1, count: Int = 2): List[Int] = {

   val n = 5

   val c = a + b
   if (count >= n) {
     List(c)
   }

   else if (a == 0 && b == 1) {
     List(a, b, c) ++ fibonacciNumbers(b, c, count + 1)
   }

   else {
     c +: fibonacciNumbers(b, c, count + 1)
   }
 }
println(RiceFields.fibonacciNumbers())

 }
```

Listing 8: Fibonacci sequence

Do the classic fibonacci sequence.

Minimal Task: Write a function that, given a dictionary, would translate a sentence. Words not found in the dictionary need not be translated.

```

     def translate(string: String): String = {
   var A: Map[String, String] = Map()

   val relatives = Map("mama" -> "mother", "papa" -> "father")

   relatives.foldLeft(string) { case (string, (key, value)) => string.replaceAll(key
  , value) }

     val line = "mama is dancing with papa"
   println("Initial: " + line)
   println("Overwritten: " + RiceFields.translate(line))


 }
```

Listing 9: Translation

Check all words in the dictionary and replace occurences.

Minimal Task: Write a function that receives as input three digits and arranges them in an order that would create the smallest possible number. Numbers cannot start with a 0.

```
      def smallest(a: Int, b: Int, c: Int): Unit = {
    var max = c

    if (a > max || b > max) {
      if (a > b)
        max = a
      else
        max = b
    }

    var min = c
    if (a < min || b < min) {
      if (a < b)
        min = a
      else
        min = b
    }

    var mid = a + b + c - min - max

    if (min != 0 && mid != 0 && max != 0)
      println(min + " " + mid + " " + max)
    else if (min == 0)
      println(mid + " " + min + " " + max)
    else if (max == 0)
      println(min + " " + max + " " + mid)
  }

      val a = 2
    val b = 4
    val c = 3
    print("Smallest order: ")
    RiceFields.smallest(a, b, c)
```

Listing 10: isPrime

Check all numbers consecutively if smaller than the next one and arrange them logically.

Minimal Task: Write a function that would rotate a list n places to the left.

rotateLeft $([1, 2, 4, 8, 4], 3) \rightarrow [8, 4, 1, 2, 4]$

```
      def rotateLeft[A](sequence: Seq[A], i: Int): Seq[A] = {
    val size = sequence.size

    sequence.drop(i % size) ++ sequence.take(i % size)
  }

    val seq: Seq[Int] = Seq(1, 2, 4, 8, 4)
    val i = 3

    println(RiceFields.rotateLeft(seq, i))
  }
```

Listing 11: Rotate Left

For example, if we have seq = Seq(1, 2, 4, 8, 4) and i = 3, then size will be 5 (the size of the sequence), and i mod size will be 3 mod 5 which equals 3. Therefore, the expression sequence.drop(i mod size) ++ sequence.take(i mod size) will evaluate to Seq(8, 4, 1, 2, 4), which represents the left-rotated sequence.

The drop(i mod size) operation will drop the first i mod size elements from the beginning of the sequence, while take(i mod size) will take the first i mod size elements from the beginning of the sequence. The two resulting sequences are then concatenated using the ++ operator, resulting in a new sequence with the desired left-rotation.

Minimal Task: Write a function that lists all tuples a, b, c such that $a^2 + b^2 = c^2$ and $a, b \leq 20$.

1 listRightAngleTriangles ()→ $[(3, 4, 5), (\ldots), \ldots.]$

```
    def listRightAngleTriangles(): List[(Int, Int, Int)] = {
  val triangles = for {
    a <- 1 until 20
    b <- 1 until 20
    c = math.sqrt(a * a + b * b).toInt
    if a * a + b * b == c * c
  } yield (a, b, c)
  triangles.toList
}
 println(RiceFields.listRightAngleTriangles())
}
```

Listing 12: Tuples

Check the input values if they fit the formula. Append to list if true.

Main Task: Write a function that eliminates consecutive duplicates in a list.

removeConsecutiveDuplicates $([1, 2, 2, 2, 4, 8, 4]) \rightarrow [1, 2, 4, 8, 4]$

```
    def consecutiveDigitsCollector(list: List[Int]): List[Int] = {
  list.head :: list.sliding(2).collect { case Seq(a, b) if a != b => b }.toList
}

    val consecutiveElementsList: List[Int] = List(1, 2, 2, 2, 4, 8, 4)
  println("Consecutive elements list: " + consecutiveElementsList)
  println("Remove occurrences: " + RiceFields.consecutiveDigitsCollector(
 consecutiveElementsList))
```

Listing 13: Duplicate Elimination

Collect all consecutive digits, remove occurences.

Main Task: Write a function that, given an array of strings, will return the words that can be typed using only one row of the letters on an English keyboard layout.

lineWords (["Hello", "Alaska", "Dad", "Peace"]) → ["Alaska", "Dad"]

```


def lineWords(words: Array[String]): Array[String] = {
  val topRow = Set('q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p')
  val midRow = Set('a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l')
  val bottomRow = Set('z', 'x', 'c', 'v', 'b', 'n', 'm')
  val rows = Array(topRow, midRow, bottomRow)

  def isOneRowWord(word: String): Boolean = {
    val wordSet = word.toLowerCase().toSet
    rows.exists(row => wordSet.subsetOf(row))
  }

  val oneRowWords = words.filter(isOneRowWord)

  oneRowWords.foreach(println)
```

```
17
18      oneRowWords
19    }
20      val listOfStrings: Array[String] = Array("Hello", "Alaska", "Dad", "Peace")
```

Listing 14: One Row Only

Check if the row contains all letters. Output the word if true.

Main Task: Create a pair of functions to encode and decode strings using the Caesar cipher.

```
1   def encryption(encrypt: String, key: Int): String = {
2     encrypt.map(c => ((c + key - 97) % 26 + 97).toChar).mkString
3   }
4
5   def decryption(decrypt: String, key: Int): String = {
6     decrypt.map(c => ((c - key - 97) % 26 + 97).toChar).mkString
7   }
8
9         val encText = "lorem"
10    val decText = "oruhp"
11    println("To be encrypted: " + encText + ". Result: " + RiceFields.encryption(
    encText, 3))
12    println("To be decrypted: " + decText + ". Result: " + RiceFields.decryption(
    decText, 3))
13
14  }
```

Listing 15: Caesar Cipher

Compute the Caesar Cipher using its algorithm and output it.

Main Task: White a function that, given a string of digits from 2 to 9 , would return all possible letter combinations that the number could represent (think phones with buttons).

lettersCombinations("23") → ["ad", "ae", "af","bd", "be", "bf", "cd", "ce", "cf"]

```
1       def combinationsOfLetters(digits: String): List[String] = {
2     val mapping = Map(
3       '2' -> "abc",
4       '3' -> "def",
5       '4' -> "ghi",
6       '5' -> "jkl",
7       '6' -> "mno",
8       '7' -> "pqrs",
9       '8' -> "tuv",
10      '9' -> "wxyz"
11    )
12
13    def generateCombinations(current: String, digits: String): List[String] = {
14      if (digits.isEmpty) List(current)
15      else {
16        for {
17          letter <- mapping(digits.head).toList
18          combination <- generateCombinations(current + letter, digits.tail)
19        } yield combination
20      }
21    }
22
23    generateCombinations("", digits)
24  }
25
```

```
26    println(RiceFields.combinationsOfLetters("23"))
```

Listing 16: Possible Combinations

The code defines a function combinationsOfLetters that takes a string of digits and returns a list of strings representing all possible combinations of letters that can be made using the letters corresponding to the given digits on a phone keypad.

The function first creates a mapping of each digit to its corresponding letters on a phone keypad using a Map.

It then defines a recursive function generateCombinations that takes two parameters - a current string representing the current combination being generated, and a digits string representing the remaining digits to be used in generating combinations.

The generateCombinations function first checks if there are no remaining digits, in which case it returns a list containing the current combination.

If there are still digits remaining, it iterates through the letters corresponding to the first digit in digits using a for loop, and recursively calls generateCombinations with the current combination appended with each of the letters iterated over, and the remaining digits excluding the first digit.

The function then returns a list of all the combinations generated by concatenating the results of each recursive call.

Finally, the combinationsOfLetters function calls generateCombinations with an empty current string and the input digits string, and returns the resulting list of combinations.

Main Task: White a function that, given an array of strings, would group the anagrams together.

```
1  def groupAnagrams(strs: Array[String]): MapView[String, List[String]] = {
2    strs.groupBy(_.sorted).mapValues(_.toList)
3  }
4
5        val strings = Array("eat", "tea", "tan", "ate", "nat", "bat")
6    val result = RiceFields.groupAnagrams(strings)
7
8  }
```

Listing 17: Anagrams

The groupAnagrams function takes an array of strings strs and returns a MapView with keys as the sorted strings in strs and values as the corresponding lists of strings from strs.

The function first calls groupBy on the input array strs, which returns a map where the keys are the elements in strs and the values are arrays of elements that are equal to the corresponding key. Here, the sorted function is used to sort the strings so that anagrams are grouped together.

The function then calls mapValues on the resulting map to convert the arrays to lists and returns the resulting MapView.

Finally, the code defines an array of strings strings and calls the groupAnagrams function on it using RiceFields as a placeholder for the object or class name that contains the groupAnagrams function. The result is assigned to result.

Bonus Task: Write a function to find the longest common prefix string amongst a list of strings.

commonPrefix([”flower”, ”flow”, ”flight”]) → ”fl”

commonPrefix ([”alpha”, ”beta”, ”gamma”]) → ””

```
  def commonPrefix(strs: List[String]): String = {
  if (strs.isEmpty) return ""
  val minLen = strs.map(_.length).min
  var i = 0
  while (i < minLen && strs.forall(_(i) == strs(0)(i))) i += 1
  strs(0).substring(0, i)
}
```

Listing 18: Longest Common Prefix

The commonPrefix function takes a list of strings strs and returns a string representing the longest common prefix that all the strings in strs share.

The function first checks if the input list is empty, in which case it returns an empty string.

Next, it finds the minimum length of strings in the list strs using map and min functions.

Then, the function initializes a variable i to 0, and enters a while loop that iterates over each character index in the strings up to the minimum length minLen. It checks if each character at index i is the same for all the strings in the input list using the forall function. If they are, the loop increments the variable i and proceeds to the next index. If they are not, the loop exits.

Finally, the function returns a substring of the first string in the input list strs(0) from index 0 up to index i. This represents the longest common prefix of all the strings in strs.

Bonus Task: Write a function to convert arabic numbers to roman numerals.

toRoman ("13") → "XIII"

```
      def arabicToRoman(arabic: String): String = {
    val arabicNum = arabic.toInt
    if (arabicNum < 1 || arabicNum > 3999) throw new IllegalArgumentException("Input
   must be between 1 and 3999")
    val numeralMap = Map(
      1000 -> "M",
      900 -> "CM",
      500 -> "D",
      400 -> "CD",
      100 -> "C",
      90 -> "XC",
      50 -> "L",
      40 -> "XL",
      10 -> "X",
      9 -> "IX",
      5 -> "V",
      4 -> "IV",
      1 -> "I"
    )
    var remaining = arabicNum
    var roman = ""
    numeralMap.keys.toList.sortWith(_ > _).foreach { key =>
      while (remaining >= key) {
        roman += numeralMap(key)
        remaining -= key
      }
    }
    roman
  }  }

      val arab = "13"
```

```
31      println(RiceFields.arabicToRoman(arab))
```

Listing 19: Arabic to Roman Numerals

Map the roman numerals to corresponding arabic ones.

Bonus Task: Write a function to calculate the prime factorization of an integer.

factorize (13) → [13] 2 factorize (42) → [2, 3, 7]

```
1     def factorize(num: Int): List[Int] = {
2     val result = scala.collection.mutable.ArrayBuffer[Int]()
3     var remaining = num
4     var factor = 2
5     while (factor <= remaining) {
6       if (remaining % factor == 0) {
7         result += factor
8         remaining = remaining / factor
9       } else {
10        factor += 1
11      }
12    }
13    result.toList
14  }  }
15
16  }
```

Listing 20: Factorization of an integer

The factorize function takes an integer num and returns a list of its prime factors.

The function first initializes an empty ArrayBuffer called result to store the prime factors.

Then, it initializes two variables remaining and factor to num and 2, respectively.

The function then enters a while loop that iterates as long as factor is less than or equal to remaining. The loop checks if remaining is divisible by factor using the modulo operator. If it is, the loop appends factor to result and updates remaining to remaining / factor, effectively reducing the number to be factorized. If it is not divisible, factor is incremented to the next prime number and the loop continues.

Finally, the function returns result as a list of prime factors.

4 P0W3 - An Actor is Born

Minimal Task: Create an actor that prints on the screen any message it receives.

```
1      import akka.actor.{Actor, ActorSystem, Props}
2
3 class Printer extends Actor {
4   override def receive: Receive = {
5     case msg: Any =>
6       println(msg)
7   }
8 }
9
10 object Print extends App {
11   //PREVIOUS WEEKS INTERACTION MOVED TO "MARKED AS DONE" CLASS
12   //week3
13
```

```
  //task 1: actor that prints any message it receives
  val system = ActorSystem("example-system")
  val printer = system.actorOf(Props[Printer], "printer")
  var input = "" //here's the input
  do {
    input = scala.io.StdIn.readLine()
    printer ! input
  } while (input != "quit") //exit command
  system.terminate()
}
  }
```

Listing 21: Print any received message

This code sets up an Akka actor system with a single actor, Printer, that prints out any message it receives. The Main object creates an instance of this actor and waits for user input. Whenever a user enters a line of text, the Printer actor receives this input as a message and prints it out. The program continues to wait for input until the user types "quit", at which point the actor system is terminated and the program ends.

Minimal Task: Create an actor that returns any message it receives, while modifying it. Infer the modification from the following example:

```

import akka.actor._
import akka.pattern.ask
import akka.util.Timeout

import scala.concurrent.Await
import scala.concurrent.duration._
import scala.io.StdIn
import scala.language.postfixOps

class MessageModifierActor extends Actor {
  def receive = {
    case i: Int =>
      val modified = i + 1
      sender() ! s"Received: $modified"
    case s: String =>
      val modified = s.toLowerCase()
      sender() ! s"Received: $modified"
    case _ =>
      sender() ! "Received: I don't know how to handle this!"
  }
}

object Main extends App {
  implicit val timeout: Timeout = Timeout(5 seconds)
  val system = ActorSystem("MessageModifierSystem")
  val actor = system.actorOf(Props[MessageModifierActor], "messageModifierActor")

  while (true) {
    val message = StdIn.readLine("Enter a message: ")
    if (message == "exit") {
      val responseFuture = actor ? message
      val response = Await.result(responseFuture, timeout.duration).asInstanceOf[
    String]
      println(response)
      system.terminate()
      sys.exit(0)
```

```
    } else {
      val responseFuture = actor ? message
      val response = Await.result(responseFuture, timeout.duration).asInstanceOf[
    String]
      println(response)
    }
  }
}

  }
```

Listing 22: Return any received message

This code sets up an Akka actor system with a single actor, MessageModifierActor, that receives messages of type Int or String, modifies them in some way, and sends back a response. The Main object creates an instance of this actor and waits for user input. Whenever a user enters a line of text, it sends that text as a message to the actor and waits for a response. If the user types "exit", the program terminates the actor system and exits. Otherwise, it prints the response and continues to wait for input.

Minimal Task: Create a two actors, actor one "monitoring" the other. If the second actor stops, actor one gets notified via a message.

```
import akka.actor._
import scala.concurrent.duration._
import scala.io.StdIn

case object Check
case class MonitorInstruction(instruction: String)
case object PrintAlive

class MonitoredActor extends Actor {
  implicit val ec = context.dispatcher
  val tick = context.system.scheduler.schedule(0.seconds, 5.seconds, self, PrintAlive
    )

  def receive = {
    case Check =>
      println("I'm still alive!")
    case MonitorInstruction(instruction) =>
      println(s"Received instruction: $instruction")
    case PrintAlive =>
      println("I'm still alive!")
  }

  override def postStop() {
    tick.cancel()
  }
}

class MonitoringActor(monitored: ActorRef) extends Actor {
  def receive = {
    case Terminated(_) =>
      println("The monitored actor has stopped!")
      context.system.terminate()
    case MonitorInstruction(instruction) =>
      monitored ! MonitorInstruction(instruction)
    case _ =>
      monitored ! Check
  }
```

```
37
38    override def preStart() {
39      context.watch(monitored)
40    }
41  }
42
43  object SupervisorTask extends App {
44    val system = ActorSystem("MonitoringSystem")
45
46    val monitoredActor = system.actorOf(Props[MonitoredActor], "monitoredActor")
47    val monitoringActor = system.actorOf(Props(new MonitoringActor(monitoredActor)), "
      monitoringActor")
48
49    while (true) {
50      val input = StdIn.readLine("Enter an instruction for the monitored actor, or '
      quit' to exit: ")
51      if (input == "quit") {
52        monitoredActor ! PoisonPill
53        system.terminate()
54        sys.exit()
55      } else {
56        monitoredActor ! MonitorInstruction(input)
57      }
58    }
59  }
60    }
```

Listing 23: Monitoring

This code defines two actor classes, MonitoredActor and MonitoringActor, and an object SupervisorTask that creates an instance of each actor and handles user input for sending messages to the MonitoredActor. The MonitoredActor periodically sends a message to itself to indicate that it is still running, and also responds to MonitorInstruction messages by printing the given instruction. The MonitoringActor watches the MonitoredActor and forwards any received messages to it, as well as terminating the system if the MonitoredActor stops running.

Minimal Task: Create an actor which receives numbers and with each request prints out the current average.

```
1       import akka.actor.{Actor, ActorLogging, ActorSystem, Props}
2
3   import scala.io.StdIn
4   import java.text.DecimalFormat
5
6   class Averager extends Actor with ActorLogging {
7     var sum: Double = 0
8     var count: Int = 0
9     val format = new DecimalFormat("#.##")
10
11    def receive: Receive = {
12      case n: Double =>
13        count += 1
14        sum += n
15        val avg = sum / count
16        val formattedAvg = format.format(avg)
17        log.info(s"Current average is $formattedAvg")
18    }
19  }
20
21  object Averager {
```

```
  def props: Props = Props[Averager]
}

object Average {
  def main(args: Array[String]): Unit = {
    val system = ActorSystem("averager-system")
    val averager = system.actorOf(Averager.props, "averager")

    while (true) {
      try {
        print("Enter a number: ")
        val n = StdIn.readDouble()
        averager ! n
      } catch {
        case _: Throwable => System.exit(0)
      }
    }
  }
}
```

Listing 24: Current Average

This code defines an Akka actor called Averager that calculates the average of numbers received through its receive method. The actor logs the current average to the console. The code also contains a main method that creates an instance of the Averager actor and repeatedly prompts the user to enter a number. The number entered is sent to the Averager actor for processing.

Main Task: Create an actor which maintains a simple FIFO queue. You should write helper functions to create an API for the user, which hides how the queue is implemented.

```
    import akka.actor._
import akka.pattern.ask
import akka.util.Timeout

import scala.concurrent.Future
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Failure, Success}
import scala.io.StdIn
import scala.language.postfixOps

case class Push(value: Any)
case object Pop
case class Popped(value: Option[Any])

class QueueActor extends Actor {
  var queue: List[Any] = Nil

  def receive = {
    case Push(value) =>
      queue = queue :+ value
      sender() ! "ok"
    case Pop =>
      val value = queue.headOption
      queue = queue.drop(1)
      sender() ! Popped(value)
  }
}

class QueueHelper {
  val system = ActorSystem("QueueSystem")
```

```
  val actor = system.actorOf(Props[QueueActor])

  def push(value: Any): Future[String] = {
    implicit val timeout = Timeout(5 seconds)
    (actor ? Push(value)).mapTo[String]
  }

  def pop(): Future[Option[Any]] = {
    implicit val timeout = Timeout(5 seconds)
    (actor ? Pop).mapTo[Popped].map(_.value)
  }

  def shutdown(): Future[Terminated] = {
    system.terminate()
  }
}

object QueueAction extends App {
  val helper = new QueueHelper()

  while (true) {
    print("Enter command (push/pop/quit): ")
    val input = StdIn.readLine()

    input match {
      case "push" =>
        print("Enter value: ")
        val value = StdIn.readLine()
        helper.push(value).onComplete {
          case Success("ok") => println("Push successful")
          case Success(_) => println("Unexpected response from server")
          case Failure(e) => println(s"Push failed with error: ${e.getMessage}")
        }
      case "pop" =>
        helper.pop().onComplete {
          case Success(Some(value)) => println(s"Popped value: $value")
          case Success(None) => println("Queue is empty")
          case Failure(e) => println(s"Pop failed with error: ${e.getMessage}")
        }
      case "quit" =>
        helper.shutdown().onComplete(_ => System.exit(0))
      case _ =>
        println("Invalid command")
    }
  }
}
```

Listing 25: Queue

This is a simple implementation of a queue using Akka actors. It defines two case classes for messages to be sent to the QueueActor actor: Push to add an element to the queue, and Pop to remove the first element from the queue. The QueueActor actor receives these messages and modifies its internal state accordingly. The QueueHelper class provides an interface to interact with the QueueActor actor, allowing the user to push and pop elements from the queue using Futures. Finally, the QueueAction object provides a simple command-line interface for the user to interact with the queue using the QueueHelper class.

Main Task: Create a module that would implement a semaphore.

```
    import akka.actor.{Actor, ActorRef, ActorSystem, Props}
```

```
import akka.pattern.ask
import akka.util.Timeout
import scala.concurrent.Await
import scala.concurrent.duration._

case object Acquire
case object Release
case object Acquired
case object SemaphoreStatus

class SemaphoreActor(var permits: Int) extends Actor {
  override def receive: Receive = {
    case Acquire =>
      if (permits > 0) {
        permits -= 1
        sender() ! Acquired
      } else {
        sender() ! false
      }
    case Release =>
      permits += 1
    case SemaphoreStatus =>
      println(s"Current number of permits available: $permits")
  }
}

object SemaphoreApp extends App {
  implicit val timeout: Timeout = 5.seconds
  val system = ActorSystem("SemaphoreSystem")
  val semaphoreActor: ActorRef = system.actorOf(Props(new SemaphoreActor(2)))

  var done = false
  while (!done) {
    print("Enter a command (acquire, release, quit): ")
    val input = scala.io.StdIn.readLine().toLowerCase
    input match {
      case "acquire" =>
        val future = semaphoreActor ? Acquire
        val result = Await.result(future, timeout.duration)
        result match {
          case Acquired =>
            println("Acquire result: true")
          case false =>
            println("Acquire result: false")
        }
        semaphoreActor ! SemaphoreStatus
      case "release" =>
        semaphoreActor ! Release
        semaphoreActor ! SemaphoreStatus
      case "quit" =>
        done = true
        system.terminate()
      case _ =>
        println("Invalid command, please try again.")
    }
  }
}
```

Listing 26: Semaphore

The program defines a SemaphoreActor class that accepts commands to acquire or release a permit,

and a SemaphoreStatus command to query the current number of permits available. The SemaphoreActor maintains an internal count of the number of permits available and responds to each command accordingly.

The main method of the program creates an instance of the SemaphoreActor with an initial count of 2 permits, and then enters a loop that reads user input commands to acquire, release, or quit. When the acquire command is received, the program sends a message to the SemaphoreActor asking to acquire a permit, and blocks until it receives a response. When the response is received, the program prints out whether the permit was acquired or not, and then sends a SemaphoreStatus command to the actor to query the current number of permits available. When the release command is received, the program sends a Release message to the SemaphoreActor to release a permit, and then sends a SemaphoreStatus command to query the current number of permits available. When the quit command is received, the program terminates the actor system and exits.

Bonus Task: Create a module that would perform some risky business. Start by creating a scheduler actor. When receiving a task to do, it will create a worker node that will perform the task. Given the nature of the task, the worker node is prone to crashes (task completion rate 50%). If the scheduler detects a crash, it will log it and restart the worker node. If the worker node finishes successfully, it should print the result.

```
    import akka.actor._
import scala.io.StdIn

object Scheduler {
  case class Task(data: String)
  case class TaskResult(result: String)
  case object WorkerCrashed

  def createScheduler(): ActorRef = {
    val system = ActorSystem("schedulerSystem")
    system.actorOf(Props[Scheduler], "scheduler")
  }
}

class Scheduler extends Actor {
  import Scheduler._

  def receive: Receive = {
    case Task(data) =>
      val worker = context.actorOf(Props[WorkerClass])
      worker ! Worker.DoTask(data)
    case TaskResult(result) =>
      println(s"Task successful: $result")
    case WorkerCrashed =>
      println("Task failed")
  }
}

class Worker extends Actor {
  import Scheduler._

  def receive: Receive = {
    case Worker.DoTask(data) =>
      if (math.random() < 0.5) {
        sender() ! WorkerCrashed
      } else {
        sender() ! TaskResult(s"Miau $data")
      }
  }
```

```
}

object Worker {
  case class DoTask(data: String)
}

object Schedule {
  def main(args: Array[String]): Unit = {
    val scheduler = Scheduler.createScheduler()

    while (true) {
      println("Enter a task to perform (or 'q' to quit):")
      val input = StdIn.readLine()

      if (input == "q") {
        println("Exiting...")
        System.exit(0)
      } else {
        scheduler ! Scheduler.Task(input)
      }
    }
  }
}
```

Listing 27: Scheduler

This is a simple example of using the Akka actor model to implement a scheduler and worker system. The Scheduler actor receives tasks and spawns a new Worker actor to perform the task. The Worker actor performs the task and sends the result back to the Scheduler. If the task fails, the Worker sends a message to the Scheduler indicating the failure. The Schedule object provides a simple command-line interface for entering tasks to be performed.

Bonus Task: Create a module that would implement a doubly linked list where each node of the list is an actor.

```
    import akka.actor._

case class Add(actor: ActorRef)
case class Next(actor: ActorRef)
case class Prev(actor: ActorRef)
case class Traverse()
case class Inverse()

class NodeActor(val value: Int) extends Actor {
  var next: Option[ActorRef] = None
  var prev: Option[ActorRef] = None

  def receive = {
    case Add(actor) =>
      next = Some(actor)
    case Next(actor) =>
      next = Some(actor)
      actor ! Prev(self)
    case Prev(actor) =>
      prev = Some(actor)
    case Traverse() =>
      var current = self
      var values = List[Int]()
      while (current != null) {
        values = values :+ current.asInstanceOf[NodeActor].value
```

```
        current = current.asInstanceOf[NodeActor].next.getOrElse(null)
      }
      println(values)
    case Inverse() =>
      var current = self
      var values = List[Int]()
      while (current != null) {
        values = values :+ current.asInstanceOf[NodeActor].value
        current = current.asInstanceOf[NodeActor].prev.getOrElse(null)
      }
      println(values)
  }
}

object DoublyLinkedListActor {
  def main(args: Array[String]) {
    val system = ActorSystem("DoublyLinkedListSystem")

    println("Enter the number of doubly linked lists to create:")
    val numLists = scala.io.StdIn.readInt()

    for (i <- 0 until numLists) {
      println(s"Enter the values for list $i, separated by spaces:")
      val values = scala.io.StdIn.readLine().split(" ").map(_.toInt)

      var prevActor: Option[ActorRef] = None
      var firstActor: Option[ActorRef] = None

      for (value <- values) {
        val nodeActor = system.actorOf(Props(new NodeActor(value)), s"NodeActor-$i-$value")

        if (prevActor.isDefined) {
          prevActor.get ! Next(nodeActor)
          nodeActor ! Prev(prevActor.get)
        } else {
          firstActor = Some(nodeActor)
        }

        prevActor = Some(nodeActor)
      }

      if (prevActor.isDefined && firstActor.isDefined) {
        prevActor.get ! Next(firstActor.get)
        firstActor.get ! Prev(prevActor.get)

        val traverseMsg = Traverse()
        firstActor.get ! traverseMsg
        val inverseMsg = Inverse()
        prevActor.get ! inverseMsg
      }
    }

    system.terminate()
  }
}
```

Listing 28: Double Linked List

5 P0W4 - The Actor is dead.. Long live the Actor

Minimal Task: Create a supervised pool of identical worker actors. The number of actors is static, given at initialization. Workers should be individually addressable. Worker actors should echo any message they receive. If an actor dies (by receiving a "kill" message), it should be restarted by the supervisor. Logging is welcome.

```
   import akka.actor._
import scala.concurrent.duration._
import scala.io.StdIn

case object Check
case class MonitorInstruction(instruction: String)
case object PrintAlive

class MonitoredActor extends Actor {
  implicit val ec = context.dispatcher
  val tick = context.system.scheduler.schedule(0.seconds, 5.seconds, self, PrintAlive
    )

  def receive = {
    case Check =>
      println("I'm still alive!")
    case MonitorInstruction(instruction) =>
      println(s"Received instruction: $instruction")
    case PrintAlive =>
      println("I'm still alive!")
  }

  override def postStop() {
    tick.cancel()
  }
}

class MonitoringActor(monitored: ActorRef) extends Actor {
  def receive = {
    case Terminated(_) =>
      println("The monitored actor has stopped!")
      context.system.terminate()
    case MonitorInstruction(instruction) =>
      monitored ! MonitorInstruction(instruction)
    case _ =>
      monitored ! Check
  }

  override def preStart() {
    context.watch(monitored)
  }
}

object SupervisorTask extends App {
  val system = ActorSystem("MonitoringSystem")

  val monitoredActor = system.actorOf(Props[MonitoredActor], "monitoredActor")
  val monitoringActor = system.actorOf(Props(new MonitoringActor(monitoredActor)), "
    monitoringActor")

  while (true) {
    val input = StdIn.readLine("Enter an instruction for the monitored actor, or '
    quit' to exit: ")
```

```
    if (input == "quit") {
      monitoredActor ! PoisonPill
      system.terminate()
      sys.exit()
    } else {
      monitoredActor ! MonitorInstruction(input)
    }
  }
}
```

Listing 29: Task Supervisor

The code defines three messages: Check, MonitorInstruction, and PrintAlive. The MonitoredActor class is an Akka actor that periodically sends the PrintAlive message to itself to indicate that it is still alive. It also responds to the Check and MonitorInstruction messages, which can be sent to it by other actors.

The MonitoringActor class is an Akka actor that watches the MonitoredActor and responds to its termination. It also forwards the MonitorInstruction and Check messages to the MonitoredActor.

Finally, the SupervisorTask object creates a MonitoredActor and a MonitoringActor, and reads input from the user to send MonitorInstruction messages to the MonitoredActor. When the user enters "quit", the MonitoredActor is stopped with the PoisonPill message, and the actor system is terminated.

Main Task: Create a supervised processing line to clean messy strings. The first worker in the line would split the string by any white spaces (similar to Python's str.split method). The second actor will lowercase all words and swap all m's and n's (you nomster!). The third actor will join back the sentence with one space between words (similar to Python's str.join method). Each worker will receive as input the previous actor's output, the last actor printing the result on screen. If any of the workers die because it encounters an error, the whole processing line needs to be restarted. Logging is welcome.

```
    import akka.actor.{Actor, ActorLogging, ActorRef, ActorSystem, PoisonPill, Props
  , SupervisorStrategy}

import scala.language.postfixOps
import scala.util.{Failure, Success, Try}

// Define messages to be sent between actors
case class CleanString(input: String)

case class SplitString(words: Array[String])

case class LowercaseAndSwap(words: Array[String])

case class JoinString(cleanedString: String)

// Define actors
class SplitStringActor(nextActor: ActorRef) extends Actor with ActorLogging {
  override def receive: Receive = {
    case CleanString(input) =>
      val words = Try(input.split("\\s+")) match {
        case Success(value) => value
        case Failure(exception) =>
          log.error(s"Error splitting string: ${exception.getMessage}")
          throw exception
      }
      nextActor ! SplitString(words)
```

```
  }
}


class LowercaseAndSwapActor(nextActor: ActorRef) extends Actor with ActorLogging {

  def switcher(arr: Array[String]): Array[String] = {
    for (i <- 0 until arr.length) {
      var str = arr(i)
      for (j <- 0 until str.length) {
        if (str(j) == 'm') {
          str = str.updated(j, 'n')
        } else if (str(j) == 'n') {
          str = str.updated(j, 'm')
        } else {
          // If the current character is neither 'n' nor 'm',
          // move on to the next character

        }
      }
      arr(i) = str
    }
    arr
  }

  override def receive: Receive = {
    case SplitString(words) =>
      //val cleanedWords =  words.map(word => word.toLowerCase().replace('m', 'n').
    replace('n', 'm'))
      val cleanedWords = switcher(words)
      nextActor ! LowercaseAndSwap(cleanedWords)
  }
}


class JoinStringActor extends Actor with ActorLogging {
  override def receive: Receive = {
    case LowercaseAndSwap(words) =>
      val cleanedString = Try(words.mkString(" ")) match {
        case Success(value) => value
        case Failure(exception) =>
          log.error(s"Error joining string: ${exception.getMessage}")
          throw exception
      }
      log.info(s"Cleaned string: $cleanedString")
  }
}

// Define supervisor strategy for actors
class StringCleaningSupervisor extends Actor with ActorLogging {

  import akka.actor.OneForOneStrategy
  import scala.concurrent.duration._

  // Restart the child actor in case of a failure
  override val supervisorStrategy = OneForOneStrategy(maxNrOfRetries = 5,
    withinTimeRange = 1 minute) {
    case _: Exception => SupervisorStrategy.Restart
  }

```

```
  // Create child actors and supervise them
  val joinStringActor = context.actorOf(Props[JoinStringActor], "joinStringActor")
  val lowercaseAndSwapActor = context.actorOf(Props(new LowercaseAndSwapActor(
   joinStringActor)), "lowercaseAndSwapActor")
  val splitStringActor = context.actorOf(Props(new SplitStringActor(
   lowercaseAndSwapActor)), "splitStringActor")

  // Send initial message to start the processing line
  override def preStart(): Unit = {
    splitStringActor ! CleanString("Messy String To Be Cleaned")
  }

  override def receive: Receive = {
    case _ =>
  }
}

// Create actor system and start supervisor actor
object StringCleaningApp extends App {
  val system = ActorSystem("StringCleaningSystem")
  val supervisor = system.actorOf(Props[StringCleaningSupervisor], "supervisor")
  Thread.sleep(1000)
  system.terminate()
}
```

Listing 30: Sstring Cleaner

The program defines four messages to be sent between actors: CleanString, SplitString, LowercaseAndSwap, and JoinString.

The program defines three actors to process these messages: SplitStringActor, LowercaseAndSwapActor, and JoinStringActor. SplitStringActor takes a CleanString message, splits the string into an array of words, and sends a SplitString message with the array to the next actor. LowercaseAndSwapActor takes a SplitString message, lowercase the words and swaps 'n' with 'm' in each word, and sends a LowercaseAndSwap message with the modified array to the next actor. JoinStringActor takes a LowercaseAndSwap message, joins the array back into a string, and logs the cleaned string.

The program defines a supervisor actor, StringCleaningSupervisor, to supervise the child actors. The supervisor strategy is set to restart the child actor in case of a failure. The supervisor actor creates the child actors and sends the initial CleanString message to the SplitStringActor.

The main method creates an actor system, creates and starts the StringCleaningSupervisor actor, waits for a second, and then terminates the actor system.

Bonus Task: Write a supervised application that would simulate a sensor system in a car. There should be sensors for each wheel, the motor, the cabin and the chassis. If any sensor dies because of a random invalid measurement, it should be restarted. If, however, the main sensor supervisor system detects multiple crashes, it should deploy the airbags. A possible supervision tree is attached below.

```
    import akka.actor.{Actor, ActorLogging, ActorRef, ActorSystem, PoisonPill, Props
  }

import scala.util.Random

case class RestartSensor(sensor: ActorRef)

case object DeployAirbags

```

```
class Sensor extends Actor with ActorLogging {
  val rand = new Random()

  override def receive: Receive = {
    case "measure" =>
      if (rand.nextInt(10) == 0) {
        // Invalid measurement, restart the sensor
        log.warning(s"${self.path.name}: Invalid measurement")
        context.parent ! RestartSensor(self)
      } else {
        // Valid measurement, send it to the supervisor
        val measurement = rand.nextInt(100)
        log.info(s"${self.path.name}: Measurement = $measurement")
        context.parent ! measurement
      }
    case _ =>
      log.warning(s"${self.path.name}: Unknown message")
  }
}

class WheelSensorSupervisor extends Actor with ActorLogging {
  val wheel1 = context.actorOf(Props[Sensor], "wheel1")
  val wheel2 = context.actorOf(Props[Sensor], "wheel2")
  val wheel3 = context.actorOf(Props[Sensor], "wheel3")
  val wheel4 = context.actorOf(Props[Sensor], "wheel4")

  var validMeasurementsCount = 0

  override def receive: Receive = {
    case "measure" =>
      wheel1 ! "measure"
      wheel2 ! "measure"
      wheel3 ! "measure"
      wheel4 ! "measure"
    case measurement: Int =>
      validMeasurementsCount += 1
      if (validMeasurementsCount == 4) {
        log.info("All wheel sensors have reported valid measurements")
        validMeasurementsCount = 0
      }
    case RestartSensor(sensor) =>
      log.warning(s"${sensor.path.name}: Restarting sensor")
      sensor ! PoisonPill
      context.actorOf(Props[Sensor], sensor.path.name)
    case _ =>
      log.warning(s"${self.path.name}: Unknown message")
  }
}

class CabinSensor extends Actor with ActorLogging {
  override def receive: Receive = {
    case "measure" =>
      val measurement = new Random().nextInt(100)
      log.info(s"${self.path.name}: Measurement = $measurement")
      context.parent ! measurement
    case RestartSensor(sensor) =>
      log.warning(s"${sensor.path.name}: Restarting sensor")
      sensor ! PoisonPill
      context.actorOf(Props[Sensor], sensor.path.name)
    case _ =>
```

```
69        log.warning(s"${self.path.name}: Unknown message")
70    }
71  }
72
73  class MotorSensor extends Actor with ActorLogging {
74    override def receive: Receive = {
75      case "measure" =>
76        val measurement = new Random().nextInt(100)
77        log.info(s"${self.path.name}: Measurement = $measurement")
78        context.parent ! measurement
79      case RestartSensor(sensor) =>
80        log.warning(s"${sensor.path.name}: Restarting sensor")
81        sensor ! PoisonPill
82        context.actorOf(Props[Sensor], sensor.path.name)
83      case _ =>
84        log.warning(s"${self.path.name}: Unknown message")
85    }
86  }
87
88  class ChassisSensor extends Actor with ActorLogging {
89    override def receive: Receive = {
90      case "measure" =>
91        val measurement = new Random().nextInt(100)
92        log.info(s"${self.path.name}: Measurement = $measurement")
93        context.parent ! measurement
94      case RestartSensor(sensor) =>
95        log.warning(s"${sensor.path.name}: Restarting sensor")
96        sensor ! PoisonPill
97        context.actorOf(Props[Sensor], sensor.path.name)
98      case _ =>
99        log.warning(s"${self.path.name}: Unknown message")
100   }
101 }
102
103 class MainSensorSupervisor extends Actor with ActorLogging {
104   val wheelSensorSupervisor = context.actorOf(Props[WheelSensorSupervisor], "
        wheelSensorSupervisor")
105   val cabinSensor = context.actorOf(Props[CabinSensor], "cabinSensor")
106   val motorSensor = context.actorOf(Props[MotorSensor], "motorSensor")
107   val chassisSensor = context.actorOf(Props[ChassisSensor], "chassisSensor")
108
109   var crashesCount = 0
110
111   override def receive: Receive = {
112     case "measure" =>
113       wheelSensorSupervisor ! "measure"
114       cabinSensor ! "measure"
115       motorSensor ! "measure"
116       chassisSensor ! "measure"
117     case RestartSensor(sensor) =>
118       log.warning(s"${sensor.path.name}: Restarting sensor")
119       sensor ! PoisonPill
120       context.actorOf(Props[Sensor], sensor.path.name)
121     case _: Int =>
122     // Valid measurement, do nothing
123     case _ =>
124       log.warning(s"${self.path.name}: Unknown message")
125   }
126 }
127
```

```
object SensorSystemExample extends App {
  val system = ActorSystem("SensorSystem")
  val supervisor = system.actorOf(Props[MainSensorSupervisor], "mainSensorSupervisor"
   )
  supervisor ! "measure"
}
```

Listing 31: Camry Sensor System

This is an example of an Akka Actor System that simulates a sensor system. The system consists of several actors that represent different types of sensors. Each sensor actor receives a "measure" message, which prompts it to generate a measurement and send it to its supervisor actor. The supervisor actor aggregates the measurements from all sensors and takes appropriate actions based on them.

The WheelSensorSupervisor actor supervises four wheel sensor actors, which simulate sensors that measure the rotation speed of the wheels. The WheelSensorSupervisor actor also keeps track of the number of valid measurements received from the wheel sensors. If all four wheel sensors report valid measurements, the WheelSensorSupervisor actor logs a message to indicate that.

The CabinSensor, MotorSensor, and ChassisSensor actors simulate sensors that measure the cabin temperature, motor temperature, and chassis temperature, respectively. These sensors generate a measurement and send it to their supervisor actor.

The MainSensorSupervisor actor supervises all other sensor actors and aggregates all measurements received from them. It also keeps track of the number of crashes that occur due to invalid measurements from the wheel sensors. If an invalid measurement is detected from a wheel sensor, the WheelSensorSupervisor actor is instructed to restart that sensor.

Finally, the SensorSystemExample object creates the actor system and starts the main sensor supervisor actor. It then sends a "measure" message to the supervisor actor to trigger the measurement process.

Bonus Task: Write an application that, in the context of actor supervision. would mimic the exchange in that scene from the movie Pulp Fiction.

```
    import akka.actor.{Actor, ActorSystem, OneForOneStrategy, Props}
import akka.actor.SupervisorStrategy._

import scala.concurrent.duration._
import scala.language.postfixOps

object PulpFictionSupervision extends App {

  // Define the messages that will be passed between actors
  case object StartConversation
  case object WhatDoesMarcellusWallaceLookLike
  case object WhatCountryAreYouFrom
  case object TheySpeakEnglishAndWhat
  case object DoYouSpeakIt
  case object SayWhatAgain
  case object DescribeMarcellusWallace
  case object DoesHeLookLikeABitch

  case object English
  case object Bald
  case object Yes

  // Define the actors
```

```
24  class Questioner extends Actor {
25    val responder = context.actorOf(Props[Responder], "responder")
26    def receive = {
27      case StartConversation =>
28        responder ! WhatDoesMarcellusWallaceLookLike
29      case WhatDoesMarcellusWallaceLookLike =>
30        println("Questioner: What does Marcellus Wallace look like?")
31        responder ! SayWhatAgain
32      case SayWhatAgain =>
33        println("Questioner: What? Say what again?")
34        responder ! DoYouSpeakIt
35      case TheySpeakEnglishAndWhat =>
36        println("Questioner: They speak English and what?")
37        responder ! English
38      case DoYouSpeakIt =>
39        println("Questioner: Say 'what' again. I dare you, I double dare you,
     motherfucker.")
40        throw new Exception("Responder: What?") // Simulate an error
41      case DescribeMarcellusWallace =>
42        println("Questioner: Describe what Marcellus Wallace looks like.")
43        responder ! Bald
44      case DoesHeLookLikeABitch =>
45        println("Questioner: Does he look like a bitch?")
46        throw new Exception("Responder: What?") // Simulate an error
47      case WhatCountryAreYouFrom =>
48        println("Questioner: What country are you from?")
49        throw new Exception("Responder: What?") // Simulate an error
50    }
51  }
52
53  class Responder extends Actor {
54    def receive = {
55      case WhatDoesMarcellusWallaceLookLike =>
56        println("Responder: He's black.")
57        sender() ! SayWhatAgain
58      case SayWhatAgain =>
59        println("Responder: He's bald.")
60        sender() ! TheySpeakEnglishAndWhat
61      case English =>
62        println("Responder: Yes.")
63        sender() ! DescribeMarcellusWallace
64      case Bald =>
65        println("Responder: He's black.")
66        sender() ! DoesHeLookLikeABitch
67      case DoesHeLookLikeABitch =>
68        println("Responder: What?")
69        throw new Exception("Questioner: I jail. You not jail.") // Simulate an error
70    }
71  }
72
73  class Supervisor extends Actor {
74    override def supervisorStrategy = OneForOneStrategy(maxNrOfRetries = 3,
     withinTimeRange = 1 minute) {
75      case _: Exception => Restart
76    }
77
78    val questioner = context.actorOf(Props[Questioner], "questioner")
79
80    def receive = {
81      case StartConversation =>
```

```
        questioner ! StartConversation
    }
  }

  // Create the system and start the conversation
  val system = ActorSystem("PulpFictionSupervision")
  val supervisor = system.actorOf(Props[Supervisor], "supervisor")
  supervisor ! StartConversation
}
```

Listing 32: Pulp Fiction

The program defines several message types that the actors will send and receive, such as "StartConversation", "WhatDoesMarcellusWallaceLookLike", "SayWhatAgain", and "TheySpeakEnglishAndWhat". These messages are used to control the flow of the conversation.

The Questioner actor initiates the conversation by sending a "StartConversation" message to the Responder actor. The Responder actor responds with the first line from the movie scene ("He's black."), and the conversation proceeds from there, with the actors taking turns sending messages to each other.

The program also defines a Supervisor actor, which supervises the Questioner actor. If the Questioner actor throws an exception, the Supervisor actor will restart it up to 3 times within a 1-minute time frame.

Overall, the program is a simple demonstration of how Akka actors can be used to simulate a conversation and handle errors in a fault-tolerant way.

6 P0W5 - May the Web be with you

Minimal Task: Write an application that would visit this link. Print out the HTTP response status code, response headers and response body.

```
object AkkaHttpClientExample extends App {
  implicit val system = ActorSystem("akka-http-client-example")
  implicit val materializer = ActorMaterializer()
  implicit val ec = system.dispatcher

  val url = "https://quotes.toscrape.com/"
  val request = HttpRequest(uri = url)

  val responseFuture: Future[HttpResponse] = Http().singleRequest(request)

  responseFuture onComplete {
    case Success(response) =>
      response.status.isSuccess() match {
        case true =>
          response.entity.toStrict(1.second).map(_.data.utf8String).foreach { body =>
            println(s"Response body: $body")
          }
        case false =>
          println(s"Request failed with status code ${response.status}")
      }
      println(s"Response headers: ${response.headers}")
    case Failure(ex) =>
      println(s"Request failed with error: ${ex.getMessage}")
  }

```

```
  // Shutdown the system after a delay
  system.scheduler.scheduleOnce(5.seconds) {
    system.terminate()
  }
}
```

Listing 33: Http Scraper

This is an example of using Akka HTTP client to send an HTTP GET request to a web server and handle the response.

First, the necessary Akka HTTP libraries are imported and an ActorSystem is created along with an ActorMaterializer and an ExecutionContext.

Next, a URL is defined and an HttpRequest object is created with the URL as its URI.

Then, an asynchronous HTTP request is sent to the server with the Http().singleRequest method, which returns a Future[HttpResponse].

Once the response is received, the onComplete method is used to handle the result. If the response status is successful, the response body is extracted and printed to the console. Otherwise, an error message is printed with the response status code. The response headers are also printed in both cases.

Finally, the ActorSystem is scheduled to be terminated after a delay of 5 seconds.

Minimal Task: Continue your previous application. Extract all quotes from the HTTP response body. Collect the author of the quote, the quote text and tags. Save the data into a list of maps, each map representing a single quote.

```
import akka.actor.ActorSystem
import akka.http.scaladsl.Http
import akka.http.scaladsl.model.{HttpRequest, HttpResponse}
import akka.http.scaladsl.model.StatusCodes._
import akka.stream.ActorMaterializer
import akka.stream.scaladsl.{Flow, Sink, Source}
import scala.concurrent.Future
import scala.util.{Failure, Success}
import spray.json._
import DefaultJsonProtocol._

case class Quote(author: String, text: String, tags: List[String])
case class QuoteResponse(contents: Map[String, List[Quote]])

object AkkaHttpClientExample2 extends App {
  implicit val system = ActorSystem("akka-http-client-example")
  implicit val materializer = ActorMaterializer()
  import system.dispatcher

  val url = "https://quotes.rest/qod.json"
  val request = HttpRequest(uri = url)

  val flow: Flow[HttpRequest, HttpResponse, Any] = Http().outgoingConnectionHttps("
    quotes.rest")

  val responseFuture: Future[HttpResponse] = Source.single(request).via(flow).runWith
    (Sink.head)

  responseFuture onComplete {
    case Success(response) =>
      response.status match {
```

```
30          case OK =>
31            response.entity.dataBytes.runFold("")((acc, curr) => acc + curr.utf8String)
    .foreach { responseBody =>
32              val json = responseBody.parseJson
33              val quotes = json.asJsObject.fields("contents")
34                .asJsObject.fields("quotes")
35                .convertTo[List[JsObject]]
36                .map { quote =>
37                  Quote(
38                    quote.fields("author").convertTo[String],
39                    quote.fields("quote").convertTo[String],
40                    quote.fields("tags").convertTo[List[String]]
41                  )
42                }
43              println(quotes)
44            }
45          case _ =>
46            println(s"Request failed with status code ${response.status}")
47        }
48        println(s"Response headers: ${response.headers}")
49      case Failure(ex) =>
50        println(s"Request failed with error: ${ex.getMessage}")
51    }
52  }
```

Listing 34: Quotes

This is another example of using the Akka HTTP client library to make an HTTP request to an external API.

In this example, the program is making a GET request to the URL "https://quotes.rest/qod.json", which is an API that provides a quote of the day. The program is using a Flow to establish a connection to the API server over HTTPS, and is then sending the HTTP request through this flow using a Source and a Sink. The runWith method is used to connect the source and sink together and run the resulting stream.

Once the response is received, the program checks the response status code. If it is a 200 (OK) status code, it extracts the response body from the response entity and parses it as JSON. It then extracts the list of quotes from the JSON and maps them to a list of Quote case class objects. Finally, the program prints the list of quotes to the console.

If the response status code is not 200, the program prints an error message to the console. In both cases, the program also prints the response headers to the console.

This program also uses Spray JSON to parse the response body as JSON and convert it to a list of Quote objects. The Quote and QuoteResponse case classes are defined at the beginning of the code.

Minimal Task: Continue your previous application. Persist the list of quotes into a file. Encode the data into JSON format. Name the file quotes.json.

```
1   import java.io.PrintWriter
2 import akka.actor.ActorSystem
3 import akka.http.scaladsl.Http
4 import akka.http.scaladsl.model.{HttpRequest, HttpResponse}
5 import akka.http.scaladsl.model.StatusCodes._
6 import akka.stream.ActorMaterializer
7 import akka.stream.scaladsl.{Flow, Sink, Source}
8 import scala.concurrent.Future
9 import scala.util.{Failure, Success}
10 import spray.json._
```

```
import DefaultJsonProtocol._

case class Quote2(author: String, text: String, tags: List[String])

object AkkaHttpClientExample3 extends App {
  implicit val system = ActorSystem("akka-http-client-example")
  implicit val materializer = ActorMaterializer()
  import system.dispatcher

  val url = "http://quotes.rest/qod.json"
  val request = HttpRequest(uri = url)

  val flow: Flow[HttpRequest, HttpResponse, Any] = Http().outgoingConnectionHttps("
    quotes.rest")

  val responseFuture: Future[HttpResponse] = Source.single(request).via(flow).runWith
    (Sink.head)

  implicit val quoteJsonFormat: JsonFormat[Quote] = jsonFormat3(Quote)
  implicit val listQuoteJsonFormat: RootJsonFormat[List[Quote]] = new RootJsonFormat[
    List[Quote]] {
    def write(list: List[Quote]): JsValue = JsArray(list.map(_.toJson).toVector)

    def read(value: JsValue): List[Quote] = value match {
      case JsArray(elements) => elements.map(_.convertTo[Quote]).toList
      case _ => throw new DeserializationException("Expected List[Quote]")
    }
  }

  responseFuture onComplete {
    case Success(response) =>
      response.status match {
        case OK =>
          response.entity.dataBytes.runFold("")((acc, curr) => acc + curr.utf8String)
    .foreach { responseBody =>
            val json = responseBody.parseJson
            val quotes = json.asJsObject.fields("contents")
              .asJsObject.fields("quotes")
              .convertTo[List[JsObject]]
              .map { quote =>
                Quote(
                  quote.fields("author").convertTo[String],
                  quote.fields("quote").convertTo[String],
                  quote.fields("tags").convertTo[List[String]]
                )
              }
            println(quotes)

            // convert the list of quotes to JSON and write it to a file
            val quotesJson = quotes.toJson(listQuoteJsonFormat)

            val writer = new PrintWriter("quotes.json")
            writer.write(quotesJson.prettyPrint)
            writer.close()
          }
        case _ =>
          println(s"Request failed with status code ${response.status}")
      }
      println(s"Response headers: ${response.headers}")
    case Failure(ex) =>
```

```
      println(s"Request failed with error: ${ex.getMessage}")
  }
}
```

Listing 35: JSON file output

Pretty same as the previous one, just create a JSON file and write everything in it.

Main Task: Write an application that would implement a Star Wars-themed RESTful API. The API should implement the following HTTP methods:

- GET /movies - GET/movies/:id

- POST /movies

- PUT /movies/:id

- PATCH /movies/: id

- DELETE/movies/:id

Use a database to persist your data. Populate the database with the following information:

```
    this code was erased due to Scala and Akka's unstable behavoir. :')
  }
```

Listing 36: isPrime

Bonus Task: Write an application that would use the Spotify API to manage user playlists. It should be able to create a new playlist, add songs to it and add custom playlist cover images. You will probably get to play with OAuth 2.0 and Base64 encoding.

```
  import SpotifyAPI.executionContext
import akka.actor.ActorSystem
import akka.http.scaladsl.Http
import akka.http.scaladsl.model._
import akka.http.scaladsl.model.headers.{Authorization, BasicHttpCredentials,
    OAuth2BearerToken}
import akka.stream.ActorMaterializer
import akka.http.scaladsl.model.Uri.Query
import akka.http.scaladsl.model.{HttpRequest, HttpResponse, StatusCodes}

import scala.concurrent.{ExecutionContextExecutor, Future}
import scala.util.{Failure, Success, Try}
import akka.util.ByteString
import akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport._
import spray.json._

import scala.concurrent.duration._
import java.net.URLEncoder
import java.nio.charset.StandardCharsets
import java.util.Base64

object SpotifyAPI {
  implicit val system: ActorSystem = ActorSystem("spotify-api")
  implicit val materializer: ActorMaterializer = ActorMaterializer()
  implicit val executionContext: ExecutionContextExecutor = system.dispatcher

  val clientId = "YOUR_CLIENT_ID"
  val clientSecret = "YOUR_CLIENT_SECRET"
  val redirectUri = "YOUR_REDIRECT_URI"

```

```
30  val scope = "playlist-modify-public"
31
32  val authUrl = s"https://accounts.spotify.com/authorize?client_id=$clientId&
     response_type=code&redirect_uri=$redirectUri&scope=$scope"
33  val tokenUrl = "https://accounts.spotify.com/api/token"
34  val apiUrl = "https://api.spotify.com/v1"
35
36  var accessToken: Option[String] = None
37
38  def authenticate(): Future[String] = {
39    val authRequest = HttpRequest(
40      method = HttpMethods.GET,
41      uri = authUrl
42    )
43
44    for {
45      authResponse <- Http().singleRequest(authRequest)
46      authCode = authResponse.uri.query().getOrElse("code", "")
47      tokenRequest: HttpRequest = HttpRequest(
48        method = HttpMethods.POST,
49        uri = tokenUrl,
50        headers = List(
51          Authorization(
52            BasicHttpCredentials(
53              Base64.getEncoder.encodeToString(s"$clientId:$clientSecret".getBytes(
     StandardCharsets.UTF_8))
54            )
55          )
56        ),
57        entity = FormData(
58          "grant_type" -> "authorization_code",
59          "code" -> authCode,
60          "redirect_uri" -> redirectUri
61        ).toEntity
62      )
63      tokenResponse: HttpResponse <- Http().singleRequest(tokenRequest)
64      tokenData <- tokenResponse.entity.dataBytes.runFold(ByteString.empty)(_ ++ _)
65    } yield {
66      accessToken = Some(tokenData.utf8String.parseJson.asJsObject.getFields("
     access_token").head.toString().replaceAll("\"", ""))
67      accessToken.get
68    }
69  }
70
71  def createPlaylist(userId: String, name: String, image: Option[String]): Future[
     String] = {
72    val createPlaylistRequest = HttpRequest(
73      method = HttpMethods.POST,
74      uri= s"$apiUrl/users/$userId/playlists",
75      headers = List(
76        Authorization(OAuth2BearerToken(accessToken.get)),
77        headers.Content-Type(ContentTypes.application/json)
78      ),
79      entity = HttpEntity(
80        ContentTypes.application/json,
81        s"""{
82           |  "name": "$name"${
83          image.map(img =>
84            s""",
85               |  "images": [
```

```
                |    {
                |      "data_uri": "$img"
                |    }
                |  ]""").getOrElse("")
        }
           |}""".stripMargin
      )
    )

    Http().singleRequest(createPlaylistRequest).flatMap { response =>
      response.status match {
        case StatusCodes.Created =>
          val result = Try(response.entity.withoutSizeLimit().dataBytes.runFold(ByteString.empty)(_ ++ _).map(_.utf8String.parseJson.asJsObject.getFields("id").head.toString.replaceAll("\"", "")))
          result match {
            case Success(value) => Future.successful(value).flatMap(identity)
            case Failure(exception) => Future.failed(exception)
          }
        case _ =>
          response.entity.toStrict(5.seconds).map(_.data.utf8String).flatMap { data =>
            Future.failed(new RuntimeException(data))
          }

      }
    }
  }

  def addTracksToPlaylist(playlistId: String, tracks: Seq[String]): Future[Unit] = {
    val trackUris = tracks.map(track => s"spotify:track:$track").mkString(",")
    val addTracksRequest = HttpRequest(
      method = HttpMethods.POST,
      uri = s"$apiUrl/playlists/$playlistId/tracks?uris=$trackUris",
      headers = List(Authorization(OAuth2BearerToken(accessToken.get)))
    )

    Http().singleRequest(addTracksRequest).flatMap { response =>
      response.status match {
        case StatusCodes.OK => Future.successful(())
        case _ =>
          response.entity.withoutSizeLimit().dataBytes.runFold(ByteString.empty)(_ ++ _).map(_.utf8String)
            .flatMap(data => Future.failed(new RuntimeException(data)))
      }
    }
  }
}

object Main {
  def main(args: Array[String]): Unit = {
    SpotifyAPI.authenticate().onComplete {
      case Success(token) =>
        println(s"Authentication successful, token: $token")

        val userId = "YOUR_USER_ID"
        val playlistName = "My Awesome Playlist"
        val tracks = Seq("TRACK_ID_1", "TRACK_ID_2", "TRACK_ID_3")
        val image = Some("YOUR_PLAYLIST_IMAGE_DATA_URI")

```

```
        SpotifyAPI.createPlaylist(userId, playlistName, image).onComplete {
          case Success(playlistId) =>
            println(s"Playlist created, id: $playlistId")

            SpotifyAPI.addTracksToPlaylist(playlistId, tracks).onComplete {
              case Success(_) =>
                println("Tracks added to playlist")
              case Failure(e) =>
                println(s"Error adding tracks to playlist: ${e.getMessage}")
            }
          case Failure(e) =>
            println(s"Error creating playlist: ${e.getMessage}")
        }
      case Failure(e) =>
        println(s"Error authenticating: ${e.getMessage}")
    }
  }
}
```

Listing 37: Spotify

In this Spotify API I've used:

- ‘akka.actor.ActorSystem‘: provides the actor system for Akka, which is used for concurrency and fault tolerance. - ‘akka.http.scaladsl.Http‘: provides HTTP client and server functionality for Akka. - ‘akka.http.scaladsl.model‘: provides HTTP model classes for Akka, such as ‘HttpRequest‘ and ‘HttpResponse‘. - ‘akka.http.scaladsl.model.headers.Authorization, OAuth2BearerToken‘: provides classes for authorization headers, such as ‘Authorization‘ and ‘OAuth2BearerToken‘. - ‘akka.stream.ActorMaterializer‘: provides an actor-based implementation of the Akka Streams API. - ‘scala.concurrent.ExecutionContextExecutor, Future‘: provides the ‘Future‘ API for concurrent programming in Scala. - ‘scala.util.Failure, Success‘: provides the ‘Try‘ API for handling exceptions and results from asynchronous computations. - ‘akka.util.ByteString‘: provides a data structure for byte strings that can be easily concatenated and split. - ‘scala.concurrent.duration‘: provides time duration classes for Scala. - ‘java.net.URLEncoder‘: provides utility methods for URL encoding. - ‘java.nio.charset.StandardCharsets‘: provides character set constants for encoding and decoding text.

As of the Spotify API object:

This is the ‘SpotifyAPI‘ object that handles the API requests and responses. Here is what each part of the object does:

- ‘implicit val system: ActorSystem‘: creates an actor system for Akka to use. - ‘implicit val materializer: ActorMaterializer‘: creates an actor-based Akka Streams materializer. - ‘implicit val executionContext: ExecutionContextExecutor‘: provides an execution context for the future-based API. - ‘val clientId‘: sets the client ID for the Spotify API. - ‘val clientSecret‘: sets the client secret for the Spotify API. - ‘val redirectUri‘: sets the redirect URI for the Spotify API. - ‘val scope = ”playlist-modify-public”‘: sets the authorization scope for the Spotify API. - ‘val authUrl‘: sets the authorization URL for the Spotify API, which includes the client ID, redirect URI, and authorization scope. - ‘val tokenUrl = ”<https://accounts.spotify.com/api/token>”‘: sets the token URL for the Spotify API. - ‘val apiUrl = ”<https://api.spotify.com/v1>”‘: sets the base URL for the Spotify API. - ‘var accessToken: Option[String] = None‘: initializes an optional access token for the Spotify API.

This is the ‘authenticate‘ method, which authenticates the application with the Spotify API using OAuth 2.0. Here is what each part of the method does:

- ‘val authRequest = HttpRequest(...)‘: creates an HTTP request to the authorization URL. - ‘for ... yield ... ‘: creates a future that chains together multiple API requests. - ‘authResponse <- Http().singleRequest(authRequest)‘: sends the authorization request and returns the response.

- ‘authCode = authResponse.uri.query().getOrElse(”code”, ””)‘: extracts the authorization code from the response URL. - ‘tokenRequest = HttpRequest(...)‘: creates an HTTP request to the token URL with the authorization code. - ‘tokenResponse <- Http().singleRequest(tokenRequest)‘: sends the token request and returns the response. - ‘tokenData <- tokenResponse.entity.dataBytes. runFold(ByteString.empty)‘: reads the token data from the response and concatenates it into a byte string. - ‘accessToken = Some(...)‘: extracts the access token from the token data and stores it in the ‘accessToken‘ variable. - ‘accessToken.get‘: returns the access token as a string.

This is the ‘createPlaylist‘ method, which creates a new playlist in the user’s Spotify account. Here is what each part of the method does:

- ‘val createPlaylistRequest = HttpRequest(...)‘: creates an HTTP request to create a new playlist with the specified name and image. - ‘method = HttpMethods.POST‘: sets the HTTP method to POST. - ‘uri = s”$apiUrl/users/$userId/playlists”‘: sets the URL to create a new playlist for the specified user ID. - ‘headers = List(...)‘: sets the authorization header and content type header for the request. - ‘Authorization(OAuth2BearerToken(accessToken.get))‘: sets the authorization header to use the access token. - ‘headers.‘Content-Type‘(ContentTypes.‘application/json‘)‘: sets the content type header to JSON. - ‘entity = HttpEntity(...)‘: sets the JSON entity for the request, which includes the name and image of the new playlist.

This is the ‘addTracksToPlaylist‘ method, which adds tracks to a playlist in the user’s Spotify account. Here is what each part of the method does:

- ‘val trackUris = tracks.map(...)‘: converts the list of track IDs to a string of track URIs.

- ‘val addTracksRequest = HttpRequest(...)‘: creates an HTTP request to add the tracks to the specified playlist.

- ‘method = HttpMethods.POST‘: sets the HTTP method to POST.

- ‘uri = s”$apiUrl/playlists/$playlistId/tracks?uris=$trackUris”‘ : $sets the URL to add the tracks to the specifie$

- ‘headers = List(Authorization(OAuth2BearerToken(accessToken.get)))‘: sets the authorization header to use the access token.

- ‘Http().singleRequest(addTracksRequest).flatMap response => ... ‘: sends the request and maps the response to a future.

- ‘case StatusCodes.OK => Future.successful(())‘: returns a successful future if the tracks were added successfully.

- ‘case _ => $Future.failed(new RuntimeException(response.entity.data.utf8String))$‘ : $returns a failed fu$

This is the ‘Main‘ object, which contains the ‘main‘ method that calls the ‘SpotifyAPI‘ methods to create a new playlist, add songs to it, and output the results. Here is what each part of the object does:

- ‘SpotifyAPI.authenticate().onComplete ... ‘: authenticates the application with the Spotify API and maps the result to a future.

- ‘case Success(token) => ...‘: if the authentication is successful, outputs the access token and sets the variables for the new playlist.

- ‘SpotifyAPI.createPlaylist(userId, playlistName, image).onComplete ... ‘: creates a new playlist in the user’s Spotify account and maps the result to a future.

- ‘case Success(playlistId) => ...‘: if the playlist is created successfully, outputs the playlist ID and adds tracks to the playlist.

- ‘SpotifyAPI.addTracksToPlaylist(playlistId, tracks).onComplete ... ‘: adds tracks to the specified playlist in the user’s Spotify account and maps the result to a future.

- ‘case Success() => ...‘: if the tracks are added successfully, outputs a success message.

- ‘case Failure(e) => ...‘: if there is an error creating the playlist or adding tracks, outputs an error message.

7 Conclusions

To sum it up concisely: In this project 0 designated to understand RTP major concepts, I’ve:

- Started to learn a new language - Scala. Further on I’m using Akka for building highly concurrent, distributed, and resilient message-driven applications.
- Understood the functional programming paradigm in which we try to bind everything in pure mathematical functions style. In addition, since Scala supports both OOP and FP (it is a concise, high-level language), it technically grants access to multiparadigm.
- Worked a lot with VCS (GitHub). Link to my repo is mentioned below.
- Implemented concurrent systems and actor models.
- Dealt with and successfully solved some Akka bugs (by using Scala actors package in a task).
- Dealt with supervisors, workers, etc., orchestrated these actors, implemented necessary and bonus functionalities.
- Did creative tasks with lots of Easter Eggs that made this project unique and intriguing to work with (Tarantino, yay!).

Finally, I’ve added my pseudonym Anna Weber to match it with my GitHub profile. A little bit of branding stuff.

References

[1] W3Schools tutorials on Scala: *https://www.w3schools.blog/scala-tutorial*

[2] Scala documentation *https://docs.scala-lang.org/overviews/core/futures.html*

[3] Akka documentation *https://akka.io/docs/*

[4] Baeldung *https://www.baeldung.com/scala/typed-akka*

[5] Stackoverflow Scala questions on various topics *https://stackoverflow.com/questions/tagged/scala*

Accessed on March 3, 2023.