Scala Programming

Basic if-else Statement

```
val x = 10
if (x > 5) {
  println(s"$x is greater than 5")
} else {
  println(s"$x is not greater than 5")
}
```

if-else as an Expression

```
val y = 3
val result = if (y % 2 == 0) "Even" else "Odd"
println(result)  // Output: Odd
```

Nested if-else

```
val z = 15
val category = if (z < 10) {
  "Small"
} else if (z < 20) {
  "Medium"
} else {
  "Large"
}
println(category)  // Output: Medium
```

Using if-else for Option Handling

```
val maybeNumber: Option[Int] = Some(4)
val numberDescription = if (maybeNumber.isDefined) {
  s"The number is ${maybeNumber.get}"
} else {
  "No number provided"
}
println(numberDescription)  // Output: The number is 4
```

if-else in Function Definitions

```
def factorial(n: Int): Int = {
  if (n <= 1) 1
  else n * factorial(n - 1)
}
```

```
println(factorial(5))  // Output: 120
```

Short-circuiting with if-else

```
val a = 10

val b = 0

val safeDivision = if (b != 0) {

  a / b

} else {

  "Cannot divide by zero"

}

println(safeDivision)  // Output: Cannot divide by zero
```

Pattern Matching with if-else

While pattern matching is generally more idiomatic in Scala, if-else statements can still be used for simple cases.

```
val age = 25

val lifeStage = if (age < 13) {

  "Child"

} else if (age < 20) {

  "Teenager"

} else if (age < 65) {

  "Adult"

} else {

  "Senior"

}

println(lifeStage)  // Output: Adult
```

Using if-else with Collections

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

val containsThree = if (numbers.contains(3)) {

  "List contains 3"

} else {

  "List does not contain 3"

}

println(containsThree)  // Output: List contains 3
```

Conditional Initialization of Variables

```
val isWeekend = true

val activity = if (isWeekend) {

  "Go hiking"
```

```scala
} else {
  "Go to work"
}
println(activity)  // Output: Go hiking
```

Complex Conditions

```scala
val temperature = 30
val humidity = 70
val weatherDescription = if (temperature > 30 && humidity > 60) {
  "Hot and humid"
} else if (temperature > 30) {
  "Hot"
} else if (humidity > 60) {
  "Humid"
} else {
  "Pleasant"
}
println(weatherDescription)  // Output: Humid
```

if-else with Type Checking

```scala
def describeType(x: Any): String = {
  if (x.isInstanceOf[Int]) {
    "This is an integer"
  } else if (x.isInstanceOf[String]) {
    "This is a string"
  } else {
    "Unknown type"
  }
}
println(describeType(42))  // Output: This is an integer
println(describeType("Scala"))  // Output: This is a string
println(describeType(3.14))  // Output: Unknown type
```

Using if-else in for-comprehensions

```scala
val mixedList = List(1, "two", 3, "four", 5)
val onlyNumbers = for (element <- mixedList if element.isInstanceOf[Int]) yield element
println(onlyNumbers)  // Output: List(1, 3, 5)
```

Scala Loops

Basic while Loop

A simple while loop that prints numbers from 1 to 5.

```
var i = 1

while (i <= 5) {

  println(i)

  i += 1

}
```

do-while Loop

A do-while loop that ensures the loop body is executed at least once.

```
var j = 1

do {

  println(j)

  j += 1

} while (j <= 5)
```

Basic for Loop

A for loop that prints numbers from 1 to 5.

```
for (k <- 1 to 5) {

  println(k)

}
```

for Loop with Range

Using a for loop with a range and a step value.

```
for (l <- 1 to 10 by 2) {

  println(l)

}
```

Iterating Over a Collection

```
val fruits = List("apples", "banana", "cherry", "oranges", "pomegranate")

for (fruit <- fruits) {

  println(fruit)

}
```

Filtering in for Loop

Using a guard to filter elements in a for loop.

```
for (m <- 1 to 10 if m % 2 == 0) {

  println(m)
}
```

Nested for Loop

Using nested for loops to iterate over a matrix.

```
val matrix = List(
  List(1, 2, 3),
  List(4, 5, 6),
  List(7, 8, 9)
)
for {
  row <- matrix
  elem <- row
} {
  println(elem)
}
```

for Loop with Yield

Using yield to create a new collection from a for loop.

```
val squares = for (n <- 1 to 5) yield n * n
println(squares)
```

for Loop Example

```
object ForLoopExample {
       def main(args: Array[String]) {
              // For Loop with Multiple Ranges
              // to : 0 - 10 & until: 2 - 9
              for ( x <- 0 to 10; z <- 2 until 10) {
                     println("Value of x :"  + x)
                     println("Value of z: " +z)
              }
       }
}
```

For loop with if condition

```
object ForLoopListExample {
       def main (args: Array[String]) {
              var rank = 0
              var list_data = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

              // for loop with Filters
              for (rank <- list_data
                     if rank < 8; if rank > 2) {
                     println("Student Rank is : " + rank)


       }
    }
}
```

Scala Functions

In Scala, functions are first-class citizens, meaning they can be passed as arguments, returned from other functions, and assigned to variables. This flexibility makes Scala a powerful language for functional programming. Here are some interesting examples demonstrating various use cases for functions in Scala.

Basic Function Definition

```
def add(a: Int, b: Int): Int = {
  a + b
}
println(add(3, 5))  // Output: 8
```

Functions without Parameters

```
def greet(): String = {
  "Hello, world!"
}
println(greet())  // Output: Hello, world!
```

Function with Default Parameters

```
def multiply(a: Int, b: Int = 2): Int = {
  a * b
}
println(multiply(4))  // Output: 8
println(multiply(4, 3))  // Output: 12
```

Anonymous Functions (Lambdas)

```
val add = (a: Int, b: Int) => a + b
println(add(3, 5))  // Output: 8
```

Higher-Order Functions

A higher-order function is a function that takes another function as a parameter or returns a function.

```
def applyFunction(f: Int => Int, value: Int): Int = {
  f(value)
}
val double = (x: Int) => x * 2
```

```scala
println(applyFunction(double, 5))  // Output: 10
```

Functions Returning Functions

```scala
def multiplier(factor: Int): Int => Int = {
  (x: Int) => x * factor
}
val triple = multiplier(3)
println(triple(5))  // Output: 15
```

Currying Functions

Currying transforms a function with multiple parameters into a series of functions each taking one parameter.

```scala
def add(a: Int)(b: Int): Int = a + b
val add5 = add(5) _
println(add5(10))  // Output: 15
```

Using Functions with Collections

```scala
val numbers = List(1, 2, 3, 4, 5)
val doubledNumbers = numbers.map(_ * 2)
println(doubledNumbers)  // Output: List(2, 4, 6, 8, 10)
val filteredNumbers = numbers.filter(_ % 2 == 0)
println(filteredNumbers)  // Output: List(2, 4)
```

Pattern Matching in Functions

```scala
def describe(x: Any): String = x match {
  case i: Int => s"Int: $i"
  case s: String => s"String: $s"
  case _ => "Unknown"
}

println(describe(42))  // Output: Int: 42
println(describe("Scala"))  // Output: String: Scala
println(describe(3.14))  // Output: Unknown
```

Recursive Functions

```scala
def factorial(n: Int): Int = {
  if (n <= 1) 1
  else n * factorial(n - 1)
}
println(factorial(5))  // Output: 120
```

Problem 1: Reverse a String

```
def reverseString(s: String): String = {
  s.reverse
}
// Example usage:
val str = "hello"
val reversedStr = reverseString(str)
println(reversedStr)  // Output: "olleh"
```

Problem 2: Find the Maximum Element in a List

Write a function to find the maximum element in a list of integers.

```
def findMax(lst: List[Int]): Int = {
  lst.max
}
// Example usage:
val numbers = List(1, 3, 5, 2, 4)
val maxNumber = findMax(numbers)
println(maxNumber)  // Output: 5
```

Problem 3: Check if a Number is Prime

Write a function to check if a given number is prime.

```
def isPrime(n: Int): Boolean = {
  if (n <= 1) return false
  for (i <- 2 until n) {
    if (n % i == 0) return false
  }
  True
}


// Example usage:
val number = 29
val isNumberPrime = isPrime(number)
println(isNumberPrime)  // Output: true
```

Problem 4: Fibonacci Series

```
def fibonacci(n: Int): List[Int] = {
  def fibHelper(x: Int, prev: Int, next: Int, acc: List[Int]): List[Int] = {
    if (x == 0) acc
    else fibHelper(x - 1, next, prev + next, acc :+ next)
  }
  fibHelper(n, 0, 1, List(0))
}

// Example usage:
val n = 10
val fibSeries = fibonacci(n)
println(fibSeries)  // Output: List(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
```

Problem 5: Sum of Elements in a List

```
def sumList(lst: List[Int]): Int = {
  lst.sum
}
// Example usage:
val numbers = List(1, 2, 3, 4, 5)
val sum = sumList(numbers)
println(sum)  // Output: 15
```

Problem 6: Palindrome Check

```
def isPalindrome(s: String): Boolean = {
  s == s.reverse
}
// Example usage:
val str = "madonna"
val isStrPalindrome = isPalindrome(str)
println(isStrPalindrome)  // Output: false
```

Problem 7: Remove Duplicates from a List

```
def removeDuplicates(lst: List[Int]): List[Int] = {
  lst.distinct
}
// Example usage:
val numbers = List(1, 2, 2, 3, 4, 4, 5)
val uniqueNumbers = removeDuplicates(numbers)
println(uniqueNumbers)  // Output: List(1, 2, 3, 4, 5)
```

Problem 8: Factorial of a Number

```
def factorial(n: Int): Int = {
  if (n == 0) 1
  else n * factorial(n - 1)
}
// Example usage:
val number = 5
val fact = factorial(number)
println(fact)  // Output: 120
```

Problem 9: Find the Length of a List

```
def listLength(lst: List[Any]): Int = {
  lst.length
}
// Example usage:
val elements = List(1, 2, 3, 4, 5)
val length = listLength(elements)
println(length)  // Output: 5
```

Using Functions with Options

```
def toInt(s: String): Option[Int] = {
  try {
    Some(s.toInt)
  } catch {
    case _: NumberFormatException => None
  }
}
val result = toInt("123").map(_ * 2)
println(result)  // Output: Some(246)

val failedResult = toInt("abc").map(_ * 2)
println(failedResult)  // Output: None
```

Step-by-Step Guide to File Analysis in Scala

```
import scala.io.Source
object FileAnalysis {
  def main(args: Array[String]): Unit = {
    val filePath = "path/to/your/file.txt"  // Update with your file path
    val source = Source.fromFile(filePath)
    val lines = source.getLines().toList
    source.close()

    // Process the file data (count words)
    val words = lines.flatMap(_.split("\\s+")).map(_.toLowerCase)
    val wordCount = words.groupBy(identity).mapValues(_.size).toSeq.sortBy(-_._2)

    // Print top 10 most frequent words
    println("Top 10 most frequent words:")
    wordCount.take(10).foreach { case (word, count) =>
      println(s"$word: $count")
    }

    // Perform analysis (average word length)
    val totalWords = words.length
    val totalChars = words.map(_.length).sum
    val averageWordLength = if (totalWords > 0) totalChars.toDouble / totalWords else 0.0

    println(s"\nTotal words: $totalWords")
    println(s"Total characters: $totalChars")
    println(f"Average word length: $averageWordLength%.2f")
  }
}
```

Object-oriented programming (OOP) in Scala

```
class Account(val accountNumber: String, var balance: Double) {
  def deposit(amount: Double): Unit = {
    if (amount > 0) {
      balance += amount
      println(s"Deposited $$amount, new balance: $$balance")
    }
  }

  def withdraw(amount: Double): Unit = {
    if (amount > 0 && amount <= balance) {
      balance -= amount
      println(s"Withdrew $$amount, new balance: $$balance")
    } else {
      println("Insufficient funds or invalid amount")
    }
  }
  def getBalance: Double = balance
}

object BankingApp {
  def main(args: Array[String]): Unit = {
    // Create instances of Account
    val account1 = new Account("12345", 1000.0)
    val account2 = new Account("67890", 2000.0)

    // Perform operations on account1
    println(s"Account number: ${account1.accountNumber}, Initial balance:
${account1.balance}")
    account1.deposit(500.0)
    account1.withdraw(200.0)
    println(s"Account number: ${account1.accountNumber}, Final balance:
${account1.getBalance}")
    // Perform operations on account2
    println(s"Account number: ${account2.accountNumber}, Initial balance:
${account2.balance}")
    account2.deposit(1000.0)
    account2.withdraw(2500.0) // Attempt to withdraw more than the balance
    account2.withdraw(1500.0) // Valid withdrawal
    println(s"Account number: ${account2.accountNumber}, Final balance:
${account2.getBalance}")
  }
}
```