Examples of Scala Workflow Functions

Workflow functions in Scala help in structuring and organizing your code, making it more readable and maintainable. Below are examples of common workflow functions in Scala, such as map, flatMap, filter, fold, reduce, and foreach, with sample code for each.

1. map

The map function transforms each element of a collection using a given function.

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

val doubled = numbers.map(_ * 2)  // List(2, 4, 6, 8, 10)


val words = List("scala", "java", "python")

val lengths = words.map(_.length)  // List(5, 4, 6)
```

2. flatMap

The flatMap function applies a function that returns a collection for each element and then flattens the result into a single collection.

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

val flatNumbers = nestedNumbers.flatMap(identity)  // List(1, 2, 3, 4, 5)


val sentences = List("Hello World", "Scala is fun")

val words = sentences.flatMap(_.split(" "))  // List("Hello", "World", "Scala", "is", "fun")
```

3. filter

The filter function selects elements of a collection that satisfy a given predicate.

```
val numbers = List(1, 2, 3, 4, 5, 6)

val evenNumbers = numbers.filter(_ % 2 == 0)  // List(2, 4, 6)


val words = List("scala", "java", "python", "ruby")

val shortWords = words.filter(_.length <= 4)  // List("java", "ruby")
```

4. fold

The fold function combines elements of a collection using a binary operation, starting with an initial value.

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

val sum = numbers.fold(0)(_ + _)  // 15
```

```
val words = List("scala", "is", "fun")

val sentence = words.fold("")((acc, word) => acc + " " + word).trim  // "scala is fun"
```

5. reduce

The reduce function combines elements of a collection using a binary operation, but does not require an initial value. The collection must be non-empty.

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

val product = numbers.reduce(_ * _)  // 120


val words = List("scala", "is", "fun")

val combinedWords = words.reduce(_ + " " + _)  // "scala is fun"
```

6. foreach

The foreach function applies a given procedure to each element of a collection.

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

numbers.foreach(println)  // Prints each number

val words = List("scala", "is", "fun")

words.foreach(word => println(word.toUpperCase))  // Prints each word in uppercase
```

7. collect

The collect function applies a partial function to the elements of a collection, returning a new collection of the results.

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

val evenNumbers = numbers.collect { case x if x % 2 == 0 => x }  // List(2, 4)

val words = List("scala", "java", "python")

val longWords = words.collect { case word if word.length > 4 => word }  // List("scala", "python")
```

8. partition

The partition function splits a collection into two collections based on a predicate.

```
val numbers = List(1, 2, 3, 4, 5, 6)

val (evenNumbers, oddNumbers) = numbers.partition(_ % 2 == 0)  // (List(2, 4, 6), List(1, 3, 5))

val words = List("scala", "java", "python", "ruby")

val (longWords, shortWords) = words.partition(_.length > 4)  // (List("scala", "python"), List("java", "ruby"))
```

9. groupBy

The groupBy function groups elements of a collection based on a given function.

```
val numbers = List(1, 2, 3, 4, 5, 6)

val groupedByParity = numbers.groupBy(_ % 2)  // Map(1 -> List(1, 3, 5), 0 -> List(2, 4, 6))

val words = List("scala", "java", "python", "ruby")

val groupedByLength = words.groupBy(_.length)  // Map(4 -> List("java", "ruby"), 5 -> List("scala"), 6 -> List("python"))
```

10. scan

The scan function is similar to fold, but it returns a collection of successive cumulative results of applying the binary operation.

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

val cumulativeSum = numbers.scan(0)(_ + _)  // List(0, 1, 3, 6, 10, 15)

val words = List("scala", "is", "fun")

val cumulativeWords = words.scan("")((acc, word) => acc + " " + word).map(_.trim)  // List("", "scala", "scala is", "scala is fun")
```

Simple Class and Object

```
// Define a class
class Car(val make: String, val model: String, var year: Int) {
  def displayInfo(): Unit = {
    println(s"Car Info: $year $make $model")
  }
}
// Create an object of the class
object SimpleClassExample extends App {
  val car = new Car("Toyota", "Corolla", 2020)
  car.displayInfo() // Output: Car Info: 2020 Toyota Corolla

  // Modify the year and display info again
  car.year = 2021
  car.displayInfo() // Output: Car Info: 2021 Toyota Corolla
}
```

Class with Companion Object

Companion objects can be used to provide factory methods and utility functions related to a class.

```
class Circle(val radius: Double) {
  def area: Double = Math.PI * radius * radius
  def circumference: Double = 2 * Math.PI * radius
}
object Circle {
  def apply(radius: Double): Circle = new Circle(radius)
}
object CompanionObjectExample extends App {
  // Create an instance using the companion object
  val circle = Circle(5.0)
  println(f"Area: ${circle.area}%.2f") // Output: Area: 78.54
  println(f"Circumference: ${circle.circumference}%.2f") // Output: Circumference: 31.42
}
```

Inheritance

This example demonstrates inheritance, where a subclass extends a superclass.

```
// Superclass
class Animal(val name: String) {
  def makeSound(): Unit = {
    println(s"$name is making a sound")
  }
}

// Subclass
class Dog(name: String) extends Animal(name) {
  override def makeSound(): Unit = {
    println(s"$name says: Woof!")
  }
}
```

```
object InheritanceExample extends App {
  val animal = new Animal("Generic Animal")
  animal.makeSound() // Output: Generic Animal is making a sound

  val dog = new Dog("Buddy")
  dog.makeSound() // Output: Buddy says: Woof!
}
```

Case Classes

- Case classes are special classes in Scala that are immutable and compared by value. They are ideal for use in pattern matching.
- Case Classes: Creating immutable data structures with built-in pattern matching support.

```
case class Person(name: String, age: Int)

object CaseClassExample extends App {
  // Create instances of case class
  val person1 = Person("Alice", 25)
  val person2 = Person("Bob", 30)

  // Print the person details
  println(person1) // Output: Person(Alice,25)
  println(person2) // Output: Person(Bob,30)

  // Copy a case class
  val person3 = person1.copy(age = 26)
  println(person3) // Output: Person(Alice,26)

  // Pattern matching with case class
  person1 match {
    case Person(name, age) => println(s"Name: $name, Age: $age") // Output: Name: Alice, Age: 25
  }
}
```

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}")
  }
}
```

Traits

Traits are similar to interfaces in Java but can also contain concrete methods.

Traits: Using traits to add multiple behaviors to classes.

```
trait Drivable {
  def drive(): Unit
}

trait Flyable {
  def fly(): Unit = {
    println("Flying in the sky!")
  }
}

class Car(val make: String, val model: String) extends Drivable {
  override def drive(): Unit = {
    println(s"Driving a $make $model")
  }
}

class Airplane extends Flyable with Drivable {
  override def drive(): Unit = {
    println("Taxiing on the runway")
  }

  override def fly(): Unit = {
    println("Flying through the clouds!")
  }
}
```

```
object TraitsExample extends App {
  val car = new Car("Honda", "Civic")
  car.drive() // Output: Driving a Honda Civic

  val airplane = new Airplane
  airplane.drive() // Output: Taxiing on the runway
  airplane.fly() // Output: Flying through the clouds!
}
```