Scala - Programming Functionally

Ayoub FAKIR
Email: ayoub.fakir@outlook.com
Twitter: @FakirSAyoub

March 1, 2019

Table of contents

Functional Programming (1)

Functional Programming

Functional Programming is expressing the fact that the programs are built exclusively using what is called *pure functions*, which means that these functions have no *side effects*. This course is constructed based on the idea that **everything is an object, literally everything**, and that almost everything we will work with will be related to **functions**. We will define what side effects are, how to avoid them and how to write functionally pure programs.

Functional Programming is a set of rules on how a program is written, not what the program will contain nor how it should interact with other components.

Functional Programming

Functional Programming aims to increase the **modularity** of our programs, and thus, their maintenability and readability.

Some of the benefits of Functional Programming include:

- Modularity
- Ease of test and use
- Better comprehension of the program's logic
- Parallelization at will!
- Programs easier to reason about (really?)

When we talk about modularity, it means that the components (functions) of the code can be understood independently of the others, and can be reused as is.

Functional Programming - Anti-Pattern

[1]

```
class Subscription {
    def buySubscription(paypalAccount: PaypalAccount,
    subPrice: Int): PaypalAccount = {
        val sub = new PaypalAccount()
        sub.paySubscription(paypalAccount, subPrice)
        /*Values are returned
        without the "return" keyword*/
        sub
    }
}
```

[1] Example based on the one given in the Red Book.

Functional Programming - No Side Effects Example

```
class Subscription {
    def buySubscription(paypalAccount: PaypalAccount,
    subPrice: Int): (PaypalAccount, ChargeSubscription)={
     val sub = new PaypalAccount()
     //Returning a tuple
     (sub,
     ChargeSubscription(paypalAccount, subPrice))
    }
}
```

Functional Programming in Scala (2)

Functional Programming in Scala

- Scala is a mix of Functional and OOP Programming.
- It runs on top of the JVM just like Java[2]!
- It is a very concise language where we can do a lot in few lines of code (to use very carefully!)
- Can operate with almost all of the Java APIs and libraries.
- Forget about the programming you learnt earlier, let's re-learn every piece of ~~functional~~ programming!

[2]and a lot better ;)

The Principle of Immutability

Immutability is the principle of *avoiding* to change the content of a variable once it is given one. Immutability goes straight with the philisophy of *no side effects* in a sense that, if it goes onto a function, its value cannot be changed.
Thus, an immutable variable, for instance, can only be used to do some calculation, and keeps the **same value**[3] and the end of the execution of a function.

[3]we're not talking about constants here!

Side Effects

In FP, a *pure function*[4] has only one job: take an input, calculate and return an output. It should not:

- Modify a variable.
- Print to the console.
- Write into a file.
- Raise an exception.
- You got the idea :)

[4] or simply *function*

Pure Functions - A Definition

Based on what we've seen until now, a *function* is one that has no side effects, in other words, a function only takes an input and produces an output.
For example, a function *f* that takes an input of type **String** and outputs something of type Int (String "arrow" Int) is a function that takes every element *s* of type String and relates it to `exactly one element` *i* of type Int.

In short, a function does nothing but calculating an output based on an input, without changing the state if the latter, and **returns exactly the same output based on the same given input**

Pure Functions - An Example

```
def lengthOfString(s: String) : Int = {
    val length = s.length()
    length
}
```

The method **lengthOfString** is a *function* that takes a variable of type String, and returns its length, a value of type Int. This method will **always** return the same value given the same string of characters as an input.

An impure function

```
def randomize(v: Int) : Int = {
    val randomValue = Math.random() + v
    randomValue
}

//Call the function
print(randomize(3)) //prints the value "9"
print(randomize(3)) //prints the value "13"
```

In this example, given the same input, the output given by the function changes; thus, this function is considered to be *non pure* or *impure*.

Higher-Order Functions

Higher-Order functions express the idea of being able to pass functions as arguments to other functions. The idea here is to consider a function as a value, because that's what it is: it does some work as a blackbox and has a value at the end of it.

These functions that we give as parameters can either be previously defined or anonymous[5]

[5] They can only be used once and are not **named functions**.

Higher-Order Functions - Examples

```scala
//If I’m 10 years older than my brother, and I’m 30,
//my brother’s age will be 20
def calculateAge(ageDifference: Int) : Int = {
    30 - ageDifference
}
def nameAndBrothersAge(name: String, ageDifference: Int,
    f: Int => Int) : (String, Int) = {
        (name, f(ageDifference))
    }

nameAndBrothersAge("Ayoub", 10, calculateAge)
```

Higher-Order Functions and Anonymous Functions

Since we began talking about HOFs, we used clearly identified functions. However, HOFs can also take *anonymous functions* as parameters (also called *function literals*, example:

```
nameAndBrothersAge("Ayoub",
    (age: Int) => age - 10)
```

Higher-Order Functions and Anonymous Functions

Let's take as an example, an *anonymous function* that aims to compare if two Strings are identical:

```
(s1: String, s2: String) => s1.equals(s2)
```

In reality, what we wrote is an object with a method called *apply*. So when creating an anonymous function, we create an object like:

```
val twoStringsAreEqual = new Function2[String,
String, Boolean] {
    def apply(s1: String, s2: String) =
        s1.equals(s2)
}
```

Now, if we call *twoStringsAreEqual*, in reality, we implicitly write *twoStringsAreEqual.apply*.

Polymorphic Functions

- Monomorphic functions are those that manage to accept only one type of Data.
- A function that operate on more than one type of data, is called a *polymorphic* function.
- Usually, polymorphic functions work for any type of data.
- A monomorphic function can also be turned into a polymorphic function if needed.

Polymorphic Functions : Signature Examples

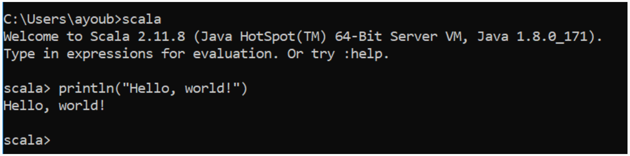
```
def zeroTheNthElement
(list: Array[String], key: String): Int



def zeroTheNthElement[A]
(list: Array[A], p: A => Boolean): Int
```

The two functions represent both a monomorphic and a polymorphic functions[6]. The first one takes an Array of String elements, and a key, and replaces the corresponding element with "zero". The second function, however, is more generic and can do the same whatever type the Array can contain.

[6]implementation in the exercises section.

Installing Scala and Running our First Program

- You can install Scala either by downloading the binaries, by installing IntelliJ or through SBT.
- https://www.scala-lang.org/download/ is the link where you can find whatever installation method you need.
- Scala comes with: 1. A compiler, and 2. A REPL (Read, Evaluate, Print, Loop) console, where you can test your Scala syntax by getting instant feedback.
- IntelliJ and SBT give you in addition the ability to create a project structure and start working with more robust Scala programs.
- You can compile your Scala class using the command **scalac HelloWorld.scala** or simply through the REP by launching the **scala** command as shown in the figure.



```
C:\Users\ayoub>scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_171).
Type in expressions for evaluation. Or try :help.

scala> println("Hello, world!")
Hello, world!

scala>
```

Scala - The Language : Expressions(3)

Variables

```
val immutableVar = "Hello Scala!"
var mutableVar = "Hello!"
```

immutableVar and *mutableVar* are both variables of type **String**. The difference is that the value of *immutableVar* can never be changed through the program, unlike *mutableVar* that can change its value at will.

We can also define a variable by giving its type explicitely like:

```
val s1: String = "Hello Scala!"
var n1: Int = 1

val errorVal: String = 3 //Throws an exception, 3 is not of type
    String.
n1 = "Hello World!" //Throws an exception, because n1 is a var
    already known as an Int, this can’t be changed.
```

When exlicitely defining the type of a variable in its declaration, if the

Type Inference

Even though Scala is a strongly typed programming language, unlike Java, we do not need to specify the type of a variable during its declaration. Once we declare a variable and we assign a value to it (this is a mandatory step in the creation of a variable as we will see later), the type is *infered*[7] from the value that the variable contains.

```
val s = "Hello!"// type: String
val n = 20// type: Int
```

[7] Can also say deduced

Types

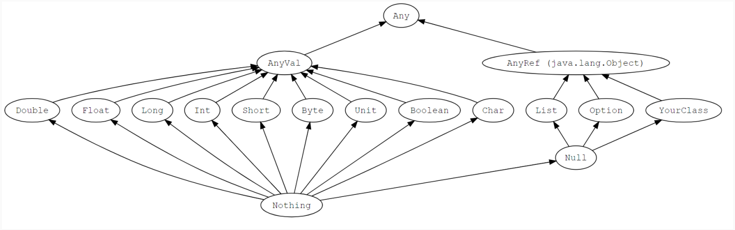


Figure 1: From Scala Docs: Types in Scala

In one hand, all types in Scala indirectly inherit from a single type: *Any*. In the other hand, these types directly inherit from either *AnyVal* or *AnyRef*[8]. If we look at the bottom of the Figure 1, we see that there is a type that inherits from all the types including Null, called *Nothing*.

[8] AnyRef is equivalent to the Object class in the Java API.

Data Immutability

It is always preferable to work with immutable Data rather than mutable data. So, in order to declare a variable, the best practice is to declare it as a *val* instead of a *var*.

This goes with the philosophy of *purity*, that is, if a variable is immutable, its state cannot be changed, and thus forces us to stay pure.

$\Rightarrow$ So avoid vars as much as possible.

String Interpolation - Examples

It's a mechanism (since Scala 2.10) for creating Strings from earlier specified data. It allows us to embed a value directly while constructing a String:

```
val age = 30
val name = "Nicolas"

val stringInter = s"The age of $name is $age"
//Output : stringInter = The age of Nicolas is 30
```

Expression Blocks

In Scala, a **Block** is a set of *one or more expressions*; thus, the value of a Block is the return value of its last expression. We will see that everything in Scala is an expression and returns something (including if/else statements and loops).

```
val age = {val birth=1990; val currentYear=2019;
    currentYear-1990}
```

The return value of the Block is the last expression, i.e. *currentYear-birth*. The Block is delimited by

```
{ ... }
```

Conditions in Scala

Just like Java or similar languages, conditions in Scala are based upon if/else statements, which are as we said earlier, expressions; **expressions in Scala return values!**, and can also be assigned to a variable for a later use.

```
if(booleanCondition) {
  //...
} else if (anotherBooleanCondition) {
  //...
} else {
  //...
}
```

Conditions and Expressions

Since a condition returns a value, it can be assigned to a variable, example:

```
val value = 3
val isEqualToZero = {if(value==0) "Zero!" else "Not Zero!"}
```

In this example, the value of **isEquelToZero** will depend on the value of **value**, it is of type *String* and can be either *"Zero!"* or *"Not Zero!"*

Introduction to Pattern Matching

Pattern Matching is a kind of an evolution of the *switch-case* in other languages like Java or C++. That said, it allows us to work with different types like Strings. So it is an expression (or value) followed by the keyword *match* and a set of *case*s encapsuled in { } .

Each case consists of a **pattern** to the left of a
$\Rightarrow$ and a result or calculation to its right.

case *pattern* $\Rightarrow$ *result*

Pattern Matching Examples

```
def matchBrand(osName: String): String = osName match {
  case "Apple" => "Wahou, a Mac System!"
  case "Microsoft" => "Who uses Windows anymore?"
  case _ => "Which Linux Distro are you using?"
}
```

The _ here means *"any other value than those defined in the* cases"

Pattern Matching Examples

Matching on case classes

```
abstract class Animal
case class Dog(name: String, race: String, age: Int,
    kindnessPercentage: Double) extends Animal
case class Cat(name: String, race: String, age: Int,
    cutenessPercentage: Double) extends Animal
```

We defined here two case classes, namely **Dog** and **Cat** that extend an abstract class **Animal**. In order to do pattern matching on those two classes we can, just as we've shown earlier do the following:

```
def whichAnimal(animal: Animal): String = {
   case Cat(name, cutenessPercentage, _) => s"The cat $name has
       is $cutenessPercentage kind!"
   case Dog(name, kindnessPercentage, _) => s"The dog $name has
       is $kindnessPercentage kind!"
}
```

Pattern Matching Examples

Pattern guards: These are simply conditions that we can apply to the left part of the arrow of a case like the following:

```
def whichAnimal(animal: Animal): String = {
    case Cat(name, cutenessPercentage, age, _) if(age>2) =>
        s"The cat $name has is $cutenessPercentage kind and is
        $age years old"
    case Dog(name, kindnessPercentage, _) if(kindnessPercentage
        < 0.4) => s"The dog $name has is not kind at all!"
}
```

In this example, the _ is used if we don't want to specify all the properties of the case classes when they're not used.

Looping!

There are different ways to use loops in Scala. The simplest form is the for (which is a short word for **for-each** in other languages). Note that a *foreach* method exists and has other usages to be see later.

```
//Create a sequence of numbers
val numbers = Seq(2, 3, 5, 3, 19)
//print the numbers sequentially:

for (n <- numbers) println(n)
//For each number "n" in the sequence "numbers", print the
    element.
```

for vs map

In order to have a good comprehension and introduction to the *map* method applied to lists in Scala, let's compare it to for. The difference between the two is quite simple: we use for if we want to iterate over the collections and print them; the map, however, is used when we want to operate a *transformation* over the list. We can basically do the same things with either fors and maps, but it's a matter of semantic. Also, the **map** method is good when dealing with immutable lists to operate transformations and "generate" transformed lists based on the input ones. **Remember, from an Input, we generate outputs after doing transformations; that's one of the joys of functional programming!**

for vs map - Example

Let's imagine we want to transform all the words of a list into uppercase words, both map and for do the job, but following different approaches and syntax:

```
val words = Seq("Hello", "World", "Bonjour", "Lemonde")
val upperCaseWordsList = new ArrayBuffer[String]()
for(word <- words) {
   upperCaseWordsList += word.toUpperCase
}/////////////////////////////////////////////////////
val upperCaseWords = words.map(word => word.toUpperCase)
upperCaseWords.foreach(println)
```

This is a very interesting example where, as we can see, the for loop didn't allow us to easily transform the list of normal words into a list of uppercase words without re-creating the list by hand; however, the map automatically generated a new list with the transformation we wanted to apply, that we assigned to a variable.

map method: A Definition

So the map method is, as we can see, a Higher-Order Function (since it takes a function as an argument) and applies the transformation (implemented in the parameter function) to each element of the list and then the result is yield to be stored in a variable and be used as we did in the previous example.

For-Comprehensions

We can also use for loops to loop through two lists at once and **yield** generators, we call those **For Comprehensions**.

```
//Create a sequence of numbers
val numbers = Seq(2, 3, 5, 3, 19)
val names = Seq("Nicolas", "Jacques", "Manu")
//yield the results and store them in a variable called result.
val result = for {
    n <- numbers
    na <- names
} yield(n, na)
```

This creates, for each element, a **pair** for each element of each sequence.

Tuples

Tuples in Scala are a set of fixed numbers of elements, each of a different type if necessary, that can either be stored in a variable to be used and useful to return multiple values from a method.

```
   val person = ("Jean", 47, "14 Rue General de Gaulle, 75002",
       "Paris")
```

The type of the variable "person" in this example is (String, Int, String, String), which is, under the hood, a shorthand for Tuple5[String, Int, String, String]. Thus, the classes that Scala uses for representing tuples are **Tuple2, Tuple3, Tuple6, ...**. Until Scala 2.10, we only could have tuples of a maximum of 23 elements. From Scala 2.11, the number of elements in a tuple can be unlimited.

Accessing Tuples

We can access tuples through their position; _1 for position one, _4 for position 4, etc... so if we do the following:

```
val address = person._3
println(address)
```

the code will print **"14 Rue General de Gaulle, 75002"** which corresponds to the address of **"Jean"**, and will be of type **String**.

Another way for accessing tuples is through **Pattern Matching**. So if we take the last example, we can go through an *implicit pattern matching*:

```
val (name, age, address, city) = person
println(name)//prints "Jean"
```

Each element of the tuple will be stored in each variable of the tuple's holder we declared, and the types will be inferred accordingly.

Functional Data Structures(4)

Pattern Matching on Types

As we've seen earlier, Pattern Matching is a powerful part of Scala allowing us to check a value against a pattern with a lot of possibilities. We can also do pattern matching on types as shown in the following code:

```
abstract class Device
    case class Phone(model: String) extends Device{
      def screenOff = "Turning screen off"
    }
    case class Computer(model: String) extends Device {
      def screenSaverOn = "Turning screen saver on..."
    }
    def goIdle(device: Device) = device match {
      case p: Phone => p.screenOff
      case c: Computer => c.screenSaverOn
    }
}
```

Code from the Scala Doc: https://bit.ly/2BIRaAz

Data Sharing

Data Sharing is the idea of being able to manipulate lists in Scala by adding or removing elements, even if those lists are *immutable*. In that regard, each time we need to remove or add an element or set of elements to a list, we create a new list; we say that we *apply transformations to the lists by giving birth to new ones.*

This allows us to return lists from methods without worrying whether they had changes that we didn't notice. In non-functional programming paradigms, we had to worry about *copying* our lists to keep their states unchanged when needed.

Lists - The Standard Scala Library

The Scala API offers us a variety of ways to create and manipulate lists, each of which can either be mutable or immutable, let's take a look to some of them:

- scala.collection.immutable.List: Used to declare linked lists immutably. It is optimal for the LIFO access pattern.
- scala.collection.Seq: It's a special case for the **Iterable** class. Random-access to Sequences through the IndexedSeq and LinearSeq subtraits.
- scala.collection.immutable.Map
- scala.collection.immutable.HashMap
- scala.collection.immutable.Iterable
- scala.collection.immutable.Queue
- scala.collection.immutable.Stream
- scala.collection.Iterator: Help iterate over a sequence of elements through the **hasNext** mainly and the **next** methods.

We'll go through the specificities of some of them in later chapters.

The folds*

The folds are operation that we can operate on Scala's collections. These folds allow us to iterate over collections by using an initial value, namely an *accumulator* and a function, which goal is to update the accumulator through each element. Here we can see some similarities with the *map* method which also applies a method to each element. The difference is that, instead of creating a new collection, the folds append the results and end up with a single value with aggregated results.

Examples

```
list.foldLeft(0) { (sum,element) => sum+element }
```

This example is a simple one in a sense that it "only" return the sum of the elements of a list, but its use is a typical one for the fold: we want to iterate over a list, append the sum of the values for each iteration, begin with an initial state and get an aggregated result. Congrats! You just defined what a fold does!

Differences between foldLeft and foldRight

The difference here is pretty simple, in a sense that, when applying the foldLeft method, the calculation goes from the left to the right (of the list), the foldRight goes from the right to the left. Note that for *associative* operations, the result doesn't change.

Trees in Scala

We've seen and worked with Lists earlier in this course. In reality, a *List* is what we call un FP an ADT: **Algebraic Data Type**. An ADT, according to the Red Book, is "SIMPLY A DATA TYPE DEFINED BY ONE OR MORE DATA CONSTRUCTORS, EACH OF WHICH MAY CONTAIN ZERO OR MORE ARGUMENTS. WE SAY THAT THE DATA TYPE OF THE *sum* OR *union* OF ITS DATA CONSTRUCTORS, AND EACH DATA CONSTRUCTOR IS THE PRODUCT OF ITS ARGUMENTS, HENCE THE NAME *algebraic* DATA TYPE." ADTs can also be used for defining data structures like a Tree:

```scala
sealed trait Tree[+A]
case class Leaf[A](value: A) extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends
    Tree[A]
```

Creating a Scala Application(5)

SBT - Simple Build Tool

Stands for Simple(Scala) Built Tool. It is interactive in a sense that it allows us to have access to a REPL for Scala through which we can access the classes of our projects. SBT is a great alternative of Maven for Scala (and even Java) projects. Unlike Maven, it allows us to define dependencies and project specificities through Scala code in a **build.sbt** file. When calling the *sbt* command in the root of a sbt folder, you access the **sbt shell** to launch commands such as *compile reload assembly* and so forth.

SBT - Next

To get started using an SBT project, you have basically two files to consider:

- build.sbt, that allows you to define your dependencies, the goals of your project and the strategies to package the whole. Rather than doing all that with some weird XML syntax (like in Maven), you can do that programmaticaly in Scala.
- assembly.sbt is the second file that allows you to define a plugin in order to create a **fat jar** out of your project sources.

build.sbt example

```
name := "scala-sample-project"
version := "0.1"
scalaVersion := "2.11.11"
resolvers += Resolver.sonatypeRepo("releases")
libraryDependencies += "com.danielasfregola" %% "twitter4s" %
    "5.5"
assemblyMergeStrategy in assembly := {
  case PathList("META-INF", xs @ _*) => MergeStrategy.discard
  case PathList("reference.conf") => MergeStrategy.concat
  case x => MergeStrategy.first
}
```

Class

Classes in Scala serve to declare objects and to define their skeleton, they form a *blueprint*. Classes in Scala can contain methods, as well as variables, constructors, traits, and even other classes. We can create an object from a class using the **new** keyword.

```scala
class Person(name: String, age: Int, address: String) {
    def getAgeAndName() : String = {
        this.name + " has " + this.age
    }

    @override def toString: String = {
        s"[ $name, $age, $address ]"
    }
}

val person = new Person("Nicolas", 30, "3 Rue Montorgueil,
    Paris")
```

Class - Constructors

```
class Person(var name: String = "Default Name", var age: Int =
    40, var address: String = "Default Address") {
}

val person = new Person //person will contain the default
    parameters
val person2 = new Person("Alain", 20, "Address") //Default
    parameters overwritten
```

In this case, we defined a class *Person* with default parameters that play the role of a constructor.

Methods

Methods in Scala are functions declared inside *classes*, *case classes* or *objects*. They can be accessed through the instanciation of the class or directly through calling the object / case class. Methods need to be **pure functions** just as we've seen throughout this course. There are few differences between methods and functions that go beyond the scope of this course. In the *Person* class we've defined, we also defined a *method* called **getAgeAndName** that returns a String corresponding to the name and the age of the person.

Case Class

A Case Class is similar to a Class apart from certain differences:

- A Case Class doesn't need the **new** keyword to be instanciated.
- Few methods come natively with the case classe like the **apply** method (which allows the case classes to be instanciated without the new keyword) or the **toString** one.
- A Case Class comes with its companion object[9]

[9] to be seen later in this course.

Object

When declaring an **Object** in Scala, in reality, we declare a **value**; thus, an object is a Singleton, a class instanciated only **once.** We declare an Object the same way we declare a class:

```
object Person(name: String, age: Int)
```

Notice that we didn't need to add the curly braces after the declaration.

Companion Object

We mean by **companion object** an object that has the same name that a Class (always in the same file), we can also say that the class is the companion class of the object. Each one of both can access to private properties and members of their companion.

Trait

A **Trait** in Scala is just similar to the **Interface** in Java. A Trait have no parameter, and can be extended by either an *Object* or a *Class*. We can use Traits as implementation *contracts* (for example, we can declare one containing methods to deal with a database, and implement it for different technologies such as MySQL, Oracle, etc...). Traits are also very useful for generic types, and their declaration and use are pretty simple:

```
trait DatabaseConnector[A] {
  def connectToDatabase: Boolean
  def listTables: List[String]
}

class MySQLDatabase(connector: MySQLConnector) extends
    DatabaseConnector[MySQLConnector] {
  override def connectToDatabase: Boolean = ???
  override def listTables: List[String] = ???
}
```

Sealed Trait

A Sealed Trait has the same definition as the Trait, the only difference is that, when declaring a Sealed Trait, the objects / classes that extend it **must** be in the same file. If the number of possible subtypes is finite (and can be controlled), use a Sealed Trait!

Abstract Class

An **Abstract Class** is a kind of a mix between the TRAIT and the CLASS. We use an abstract class when we want, for instance, a similar behavior than a Trait but with a parameter, example:

```
trait Person(name: String)//ERROR!
abstract class Person(name: String)//OK!
```

The Abstract Class can be extended just as we do with a Trait. Also, another reason is that the Trait cannot be called from a Java Code (for reasons of compatibility); an Abstract Class can.

Errors Handling(6)

Handling Errors Functionally

Sure enough, from the beginning, the statement is that in functional programming, we write *pure functions*, avoiding *side effects*. Thus, throwing an exception is considered as a side effect and should be handled differently. The idea that we will be defending in this section is that exceptions and errors need to be handled just as they were like any other value, and write code that behaves accordingly.

It is advised from the **Red Book** that two types need to be used to handle errors and exceptions: **Option and Either**; we will follow the rule and use them through this section as well.

Errors are just... A State!

The idea that we try to express here is that exceptions should not only be used for error handling... But to be part of the control flow and the execution of our application.

Here we want to consolidate error-handling logic by applying the following simple principle: instead of throwing an exception, we return a **value**, that says that the code behaved unexpectedly through the state of its input.

Functional Exceptions: How and when to use them?

Let's take the following example:

```scala
def sumValuesOfList(s: Seq[Int]): Int = {
    if(s.isEmpty) {
        throw new ArithmeticException("Cannot perform sum on an
            empty sequence")
    }
    s.foldLeft(0)(_ + _)
}
```

The **sumValuesOfList** is called a *partial function*, meaning that it is not defined for some inputs (an empty list for instance). In Scala, it is not the right way of throwing exceptions, we could do better!

Handling Exceptions in Scala

If we take the last example, we can manage the empty list scenario differently like:

```
  def sumValuesOfList(s: Seq[Int]): Int = {
    if(s.isEmpty) {
        0
    }
    s.foldLeft(0)(_ + _)
}
```

This way, rather than stopping the program (especially if the method is used more than once for several lists), we can return a **value** asserting that the calculation didn't succeed.

Handling Exceptions in Scala

Another solution might be the one that forces the caller of a function to provide a value to be returned in case of error:

```
    def sumValuesOfList(s: Seq[Int], onError: Int): Int = {
    if(s.isEmpty) onError
    s.foldLeft(0)(_ + _)
}
```

Here the caller has to know *exactly* what her function returns and how to manage errors.

Never Null... Always Option!

We'll begin talking about the *Option* data type by agreeing on one thing: AVOID NULL when trying to manage errors and exceptions! When declaring a variable as an Option of some type:

```
val name: Option[String]...
```

This means that the variable **name** can either contain a result: **Some(someString)** or nothing: **None**. So if we take again the previous example and try to manage the emptiness of our list with the Option data type, that would mean:

Handling errors with an Option

```scala
def sumValuesOfList(s: Seq[Int]): Option[Int] = {
    if(s.isEmpty) None
    Some(s.foldLeft(0)(_ + _))
}
```

Meaning: We can have a value, namely **Some**, or nothing, namely **None**. We say that **sumValuesOfList** is a *total function*.

Tea or Coffee? - The Either data type in Scala

The **Either** data type is an extension to the **Option** one we've seen earlier. Its definition is the following:

```scala
sealed trait Either[+E, +A]
case class Left[+E](value: E) extends Either(E, Nothing)
case class Right[+A](value: A) extends Either(Nothing, A)
```

The similarity here between Either and Option is that both have only two cases; the difference is that in Either, in both cases, we carry an information, a *value*.

Handling errors with an Either

If we take the same example as earlier, we can have the following implementation:

```
def sumValuesOfList(s: Seq[Int]): Either[String, Int] = {
    if(s.isEmpty)
        Left("Cannot perform sum operation on an empty list")
    else
        Right(s.foldLeft(0)(_ + _))
}
```

Here, we have a message if the operation cannot be performed. This **Left** part can contain anything we want, even an exception like the following:

Handling errors with an Either

```
def safeDiv(x: Int, y: Int): Either[Exception, Int] = {
    try Right(x / y)
    catch { case e: Exception => Left(e) }
}
```

The last example comes from the Red Book.

Expressions In More Depth(7)

Look over data: scan and reduce

These two methods, namely **scan** and **reduce** will be easier to understand since we've seen the **fold** method in action. Basically the new two methods that we introduce today work the same:

- The *reduce* method folds exactly the same way as *fold*, and we do not need to give it an initial value.
- The *scan* methods does the same as well, but its return value is not an aggregated single value; instead, it returns a **collection** of intermediate cumulative results, using a start value just as the fold method.

The two methods have also two ways to traverse the lists: from the right or from the left: *scanLeft, scanRight, reduceLeft, reduceRight*
Implementations go as follows:

```
val list = List(4, 5, 8, 9)
list.reduce(_ + _) //returns: 26
list.scan(0)(_ + _)//returns: List(0, 4, 9, 17, 26)
```

Advanced Topics in Scala(8)

What do we mean by Composition?

In Scala, *compose* makes a new function based on two other ones: "f(g(x))"[10], can be noted $f \circ g$ as well. In Scala, we can achieve composition through two functions: *compose* and *andThen*.

```
def f(s: String) = "f(" + s + ")"
def g(s: String) = "g(" + s + ")"
val fComposeG = f _ compose g _
print(fComposeG("School"))//returns: f(g("School"))
val fAndThenG = f _ andThen g _
print(fAndThenG("School"))//returns: g(f("School"))
```

In this syntax, *"f _"* means that we are turning the method "f" into a function, since both functions *compose* and *andThen* only take functions as parameters, not methods. The difference then between the two composing methods is the order in which the *f and g* are applied.

[10] we'll see more on that when talking about monoids.

Purely Functional State

The idea of "Purely Functional State" is basic: using pure functions, as we defined them, that take inputs, produce outputs and take states in their arguments, returning a new state alongside with its result. Now the core idea of PFS is to be able to deal with functions that have side effects and turn them into pure functions. Let's take, as an example, the *append* function from the StringBuilder API. This method has side effects because if we call it twice on the same input, it won't have the same output. Let's try to turn this method to its PFS equivalent:

```
trait StringBuilderAppender {
    def appendString(appendedString: String): (String,
        String)
}
```

Here, instead of returning only the result of the append, we also return the new state, while the old one will remain unmodified.

Purely Functional State

The idea here is to decouple the *next state* from the way we *communicate* the result to the rest of the program.

```
case class SBA(initialString: String) extends
     StringBuilderAppender {
    def appendString(s: String) : (String, String) = {
        val stringBuilder = new StringBuilder(initialString)
        val result = new
             StringBuilder().append(initialString).append(s)
        (result.toString, initialString)
    }
}
```

Purely Functional State

```
val sba = SBA("Hello, ")
    val (result, state) = sba.appendString("World!")
    println(result)//result will contain "Hello, World!"
    val (result1, state1) = sba.appendString("World!")//result1
        will contain "Hello, World!"
```

In other words, whenever we have the same input, the same output is produced, hence, our appendString is a *pure function*

Purely Functional Parallelism

-

Tails

The Tail function in Scala is is an optimized way to do recursion in functional programming. The tail method (or annotation) returns a collection consisting of all elements except the first one. In that regard, Scala provides an annotation, namely **TailRec** to do recursive calculations in the most efficient way possible; when writing a recursive function, if we annotate it with @TailRec, the compiler will make it a priority to optimize it in the heap.

Strictness and Laziness

A Non-Strict function is a function that always evaluate its arguments. That's how most of the other programming languages work. In Scala, we can choose to make a function's arguments (or the function itself) lazy: that would mean that the function won't be evaluated unless we explicitely call it.

```
val number1 = 4 * 5//strict
val number2 = 5 / 2//strict
lazy val number3 = 4 + 3//lazy
lazy val number4 = number3 + number2 //lazy
lazy val number5 = 9 * 10//lazy
```

Monoids

Even if the term can seem a little hard to apprehend, a monoid is a simple structure in the Category Theory of mathematics; in the latter, it means a category with one object. Monoids are very useful in Functional Programming, and we used them throughout the course without even noticing! Some of their benefits include:

- Facilitating parallel computation by breaking down a significant problem into chunks to be calculated separately.
- Assembling simple pieces to form complex calculations.

Monoids - A definition

So how can we define this thing, *Monoid*? Simply put, a monoid obeys to some rules, or more precisely, **laws**, three, that should always be respected:

- A type, whatever it can be, let's call it B.
- An associative binary operation (think of + or * as associative operations, since (a + b) + c is strictly equal to a + (b + c), same goes with the * operator).
- The identity, that we call the *zero* value. For instance, this *zero* value is *0* for the + operator, *1* for the * operator, "" for Strings and *Nil* for Lists.

To sum up, a Monoid is a type *B* and an implementation of *Monoid[A]* that satisfies the three laws. As stated in the Red Book, *a monoid is a type together with a binary operation over that type, satisfying associativity and having an identity element*.

Monoids and Lists

Let's do it in terms of a simple example: creating a Monoid and use a FoldLeft function on top of it:

```
val books = List("The Laws of Success", "Think and Grow
    Rich", "Functional Programming in Scala")
books.foldRight(myMonoid.zero)(myMonoid.operator)
```

Note here that it doesn't matter whether we use foldRight or foldLeft, since the associativity property is respected in the monoids, so we should have the same result: **The Laws of SuccessThink and Grow RichFunctional Programming in Scala**

Monads

-

Functors

-

Monads vs Applicative Functors

-

Traversable Functors

-

Scoping the Side Effects.

-

Stream Processing in Scala

-

Exercices

Exercices

-