[Scala - The Diatonic Syallable](https://play.fresco.me/course/255)

Course Introduction

In this course, you will learn some of the advanced Scala concepts such as:

* **Implicits**
* **For Expressions**
* Functional concepts - **Currying** and **Tail Recursion**
* **Options**
* Concurrency - **Parallel Collections** and **Actors**

##### Implicits

Implicits is an interesting feature which simplifies your code by leaving out obvious details which the compiler can automatically infer. There are two ways to use implicits:

• An **implicit parameter** is a parameter to a method which doesn't have to be passed explicitly. The value of the parameter is inferred by the Scala compiler based on its type and the implicit values in scope.

• An **implicit conversion** is a function that converts one type to another. This function is called by the Scala compiler when a type conversion is required.

##### Implicit parameters

Following is a method ***find*** defined with an implicit parameter ***x***. The keyword **implicit** is used to mark an implicit parameter.

`def find(implicit x : Int) = x

`

You can call this method like any other method by passing an integer value for the parameter ***x***.

find(10) will return value **10**.

Now you can call the same method without passing any parameter, if you have an implicit integer value defined in scope as given below.

implicit val test = 5

find //returns the value 5

The scala compiler passes the implicit integer value ***test*** to the ***find*** method when you call it without any parameter. So you get **5** as the function return value.

**Code:**

**Instructions:**

* Write a function muliply. Which takes two integer variable x and y and returns an ouput in which y is an implicit parameter, and also declares the implicit parameter in the function as 5.

import java.io.\_

import java.math.\_

import java.security.\_

import java.text.\_

import java.util.\_

import java.util.concurrent.\_

import java.util.function.\_

import java.util.regex.\_

import java.util.stream.\_

import scala.collection.immutable.\_

import scala.collection.mutable.\_

import scala.collection.concurrent.\_

import scala.concurrent.\_

import scala.io.\_

import scala.math.\_

import scala.sys.\_

import scala.util.matching.\_

import scala.reflect.\_

object Result {

    /\*

     \* Complete the 'multiply' function below.

     \*

     \* The function is expected to return an INTEGER.

     \* The function accepts INTEGER x as parameter.

     \*/

    def multiply(x: Int)(implicit y: Int = 5): Int = return x\*y// Put your code here

}

object Solution {

    def main(args: Array[String]) {

        val printWriter = new PrintWriter(sys.env("OUTPUT\_PATH"))

        val x = StdIn.readLine.trim.toInt

        val result = Result.multiply(x)

        printWriter.println(result)

        printWriter.close()

    }

}

**Code:**

**Instructions:**

* Write an implicit function which takes a string as a parameter and converts it to a CustomString
* Follow the instructions in the comments to proceed with the handson.

import java.io.\_

import java.math.\_

import java.security.\_

import java.text.\_

import java.util.\_

import java.util.concurrent.\_

import java.util.function.\_

import java.util.regex.\_

import java.util.stream.\_

import scala.collection.immutable.\_

import scala.collection.mutable.\_

import scala.collection.concurrent.\_

import scala.concurrent.\_

import scala.io.\_

import scala.math.\_

import scala.sys.\_

import scala.util.matching.\_

import scala.reflect.\_

object Result {

//  Define class customString which accpets a string

//  Define a method isNumeric inside this class to check if each digit in the number is of numeric type or not

// Define a implicit function strToCustomString which accepts a string and returns a object of type CustomString

    class customString(str1:String) {

        def isNumeric = str1 forall Character.isDigit

    }

    implicit def strToCustomString(s: String): customString = new customString(s)

// Avoid handling the test method.

def test(var1:String){

    val output = var1.isNumeric

    println(output)

}

}

object Solution {

    def main(args: Array[String]) {

        val test1 = StdIn.readLine

        Result.test(test1)

    }

}

Input (stdin)

* **100**

Your Output (stdout)

* **true**

**code:**

1. **Multiple Implicit Conversions:**

**Instructions:**

* Write an implicit function which takes a string as a parameter and converts it to a CustomString.
* Write another implicit function which takes a string and converts it to CustomInt.

import java.io.\_

import java.math.\_

import java.security.\_

import java.text.\_

import java.util.\_

import java.util.concurrent.\_

import java.util.function.\_

import java.util.regex.\_

import java.util.stream.\_

import scala.collection.immutable.\_

import scala.collection.mutable.\_

import scala.collection.concurrent.\_

import scala.concurrent.\_

import scala.io.\_

import scala.math.\_

import scala.sys.\_

import scala.util.matching.\_

import scala.reflect.\_

object Result {

    class CustomString(val s: String) {

        // Define method with name findLength to get the length of the string 's'

        def findLength = s.length

    }

    class CustomInt(val number : Int){

        //Define method with name multiply to perform multiplication of the 'number' by 10

        def multiply = number \* 10

    }

    // Define your implicit functions here.

    implicit def strToCustomString(str1: String): CustomString = new CustomString(str1)

    implicit def strToCustomInt(str2: String): CustomInt = new CustomInt(str2.toInt)

    // Don't modify the test method.

    def test(var1:String){

        val o1 = var1.findLength

        println(o1)

        val output = var1.multiply

        println(output)

    }

}

object Solution {

    def main(args: Array[String]) {

        val test1 = StdIn.readLine.trim

        Result.test(test1)

    }

}

Input (stdin)

* **100**

Your Output (stdout)

* **3**
* **1000**

What is the output of the following code ?

object PrintHello extends App{

def print(x: Int) = println(x)

implicit def strToInt(x : String) = x.length

print("hello")

}

What is the output of the following code ?

object PrintHello extends App{

def print(x: Int) = println(x)

print("hello")

}

type mismatch; found : String("hello") required: Int

The value of the implicit parameter is inferred by the Scala compiler based on its type and the implicit values in scope.

True

##### For Comprehensions

Generally, a for-comprehension is of the form:

**for (seq) yield expr**

The **seq** is a sequence of **generators**, **variable definitions** and **filters**. For example,

for {

p <- persons // a generator

n = p.name // a definition

if (n startsWith "T") // a filter

} yield n

Also there can be multiple generators separated by semicolon as shown here:

for (i <- list1; j <- list2)

yield (i, j)

##### For Comprehensions

Here are two simple for-comprehension examples which return two Lists.

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

for (i <- list)

yield i+10

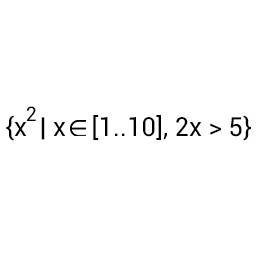
// Output is List(11,12,13,14,15)

for (i <- list if i % 2 == 0)

yield i

// Output is List(2,4)

##### For Comprehensions



For-expression corresponding to the equation given in the image is

for(x <- 1 to 10 if 2\*x > 5)

yield x\*x

//returns Vector(9, 16, 25, 36, 49, 64, 81, 100)

What is the output of the following ?

object Main extends App {

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

val newList = for (i <- list) yield i\*i

println(newList)

}

Question Type: **Single-Select**

Complete the code to filter even numbers from given List(1, 2, 3, 4, 5, 6). Expected output is List(2, 4, 6)

object Main extends App {

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

// define newList here

// val newList =

println(newList)

}

val newList = for (i <- list if i%2==0) yield i

What is the output of the following ?

object Main extends App {

case class Student(val name: String, val score: Int)

val studentList = List(new Student("John", 9),

new Student("Thomas", 5),

new Student("Jennifer", 7),

new Student("Lora", 8))

val topScorers = for (student <- studentList if (student.score >8))

yield student.name

topScorers.foreach(name => println(name))

}

John

##### Options

Suppose you have a method which returns a string value optionally, i.e.. this method returns a valid string in most of the cases, but in some special cases, there is no string value to return.

If you are working in other programming languages like Java, you would have returned a **null** in such a case. But the client code, which calls your method, may not be expecting **null** as the return value. If the null type is not handled in the client code it will result in a **NullPointerException**.

In Scala, you don't return null values. Instead, you make it explicit to the client code that the return value of your method is an optional string. This is done using the **Option[A]**trait.

##### Options



Option can be considered as a container which can hold values of any type. Such a value can be of two forms: **Some(x)**, where **x** is the actual value, or the **None** object, which represents a missing value.

Following is a sample Scala method which returns a String value optionally. Note that the return type is **Option[String]** instead of **String**.

def find(x : Int) : Option[String] = {

…// some implementation

}

##### Options examples

If you have to return Integer value optionally, the return type would be **Option[Integer]**. The client code will call the find method and handle the return value as shown.

val result = find(10)

if(result.isDefined){

println(result.get) //handle string value

}

else {

println(“no value”) //handle empty value

}

where **result** is of type **Option[String]**. You can use **result.get** to get the string value, after checking for content using the **isDefined** method.

Another way to handle result is using a **pattern matching** block.

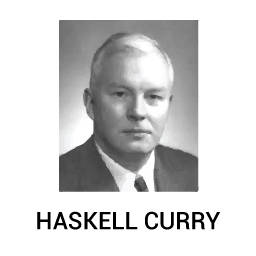
result match {

case Some(x) => println(x) //handle the string value

case None => println(“no value”) //handle empty value

}

##### Currying



**Currying** is a technique to transform a function that takes multiple arguments in such a way that it can be called as a **chain of functions**, each with a single argument.

Following is a regular function with multiple parameters:

def add(a: Int, b: Int) : Int = a + b

You can convert this to a **curried function** by simply separating the parameters into multiple parameter lists.

def add(a: Int)(b: Int) : Int = a + b

This technique got its name from the American mathematician, **Haskell Curry**.

##### Calling a curried function

You can call a curried function by passing all the parameters:

def add(a: Int)(b: Int) : Int = a + b

add(1)(2) //returns 3 :Int

Or by passing fewer parameters (partial application).

add(1) \_ //return scala.Function1

In the second case you get back a function instead of an integer.

def add(a: Int)(b: Int) : Int = a + b is actually a shorthand for the following:

def add(a: Int) : (Int => Int) = {

(b :Int) => a + b

d

**code:**

**Instructions:**

* Write the function show which uses the Option to return the capital of a country.
* Return the capital of the country if the name of the country exists in the Map provided.
* Return?, if there is no country name in the Map

import java.io.\_

import java.math.\_

import java.security.\_

import java.text.\_

import java.util.concurrent.\_

import java.util.function.\_

import java.util.regex.\_

import java.util.stream.\_

import scala.collection.mutable.\_

import scala.collection.concurrent.\_

import scala.concurrent.\_

import scala.io.\_

import scala.math.\_

import scala.sys.\_

import scala.util.matching.\_

import scala.reflect.\_

object Result {

  // Define your show function here

  def show(x: Option[String]) = {

      x match {

          case Some(s) => s

          case None => "?"

      }

  }

}

object Solution {

    def main(args: Array[String]) {

        val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo")

        println("Capital of Japan : " + Result.show(capitals.get( "Japan")) )

        println("Capital of India : " + Result.show(capitals.get( "India")) )

    }

}

A powerful Scala idiom is to use the \_\_\_\_\_ class when returning a value from a function that can be null.

Option

What is the output of the following ?

object Example extends App {

val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo")

println("Japan: " + show(capitals.get( "Japan")) )

def show(x: Option[String]) = x match {

case Some(s) => s

case None => "?"

}

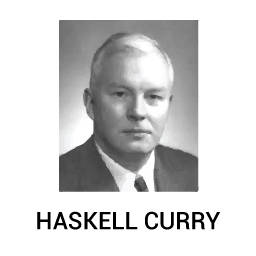
}

* 1. Japan: Tokyo

The Option object is either an instance of the Scala Some class or None class.

True

##### Currying



**Currying** is a technique to transform a function that takes multiple arguments in such a way that it can be called as a **chain of functions**, each with a single argument.

Following is a regular function with multiple parameters:

def add(a: Int, b: Int) : Int = a + b

You can convert this to a **curried function** by simply separating the parameters into multiple parameter lists.

def add(a: Int)(b: Int) : Int = a + b

This technique got its name from the American mathematician, **Haskell Curry**.

What is the output of the following code snippet ?

def concatenator(w1: String)(w2: String) = w1 + " " + w2

println(concatenator("Hey")("Currying"))

* 1. Hey Currying

What is the output of the folowing ?

object CurryingExample extends App{

def concatenator(w1: String)(w2: String) = w1 + " " + w2

val heyWord = concatenator("Currying")\_

println(heyWord("Demo"))

}

Currying Demo

Code:

Instructions:

* Write a partial function customMod from modNum to get the output as input1 % input2

object Result {

    /\*

     \* Complete the 'customMod' function below.

     \*

     \*/

    def partialFunctionImplementation(input1: Int, input2: Int) {

        def modNum(input1: Int)(input2: Int) = input1 % input2

        //Put your code here to create the partial function.

        val customMod = modNum(input1)\_

        val output = customMod(input2)

        println(output)

    }

}

object Solution {

    def main(args: Array[String]) {

        val input1 = StdIn.readLine.trim.toInt

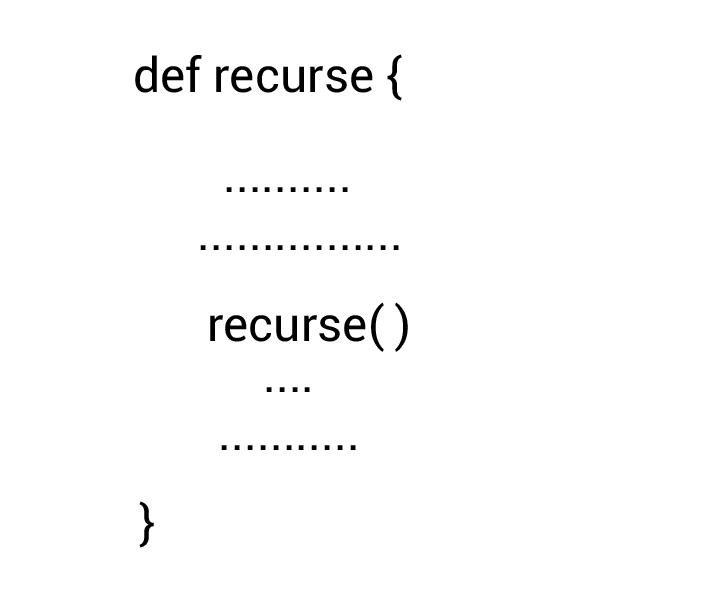
        val input2 = StdIn.readLine.trim.toInt

        Result.partialFunctionImplementation(input1, input2)

    }

}

##### Tail Recursion



A function which calls itself is known as a **recursive** function. A recursive function becomes **tail recursive** when the recursive call is the last thing executed by the function. The following function finds the factorial of a given number.

def factorial(n: Int): Int =

if (n == 0) 1

else n \* factorial(n – 1)

Here the **factorial** function makes a call to itself and hence is a recursive function, but not a tail recursive function since the last step here is multiplication operation **n \* factorial(n – 1)**. Since **factorial(n – 1)** is one of the operands, it is evaluated before multiplying.

##### Tail Recursion

A modified version of the factorial function which is tail recursive is as follows:

def factorial(x: Int): Int = {

def iter(n: Int, acc: Int): Int =

if (n == 0) acc else iter(n-1, n \* acc)

iter(x, 1)

}

**Tail recursion** is important because it is implemented more efficiently than the general recursion.

If a recursive function is tail recursive, the SCALA compiler will convert it to a **loop**. It will only use a single stack frame and executes just as efficiently as a loop.

The compiler can check for tail-recursion using a **@tailrec annotation.**

@tailrec

def factorial(x: Int): Int = ...

If the annotation is given, an error would be thrown if the implementation of factorial was not tail recursive.

What is the output of function factorial ?

@tailrec

def factorial(n: Int): Int = if (n == 0) 1 else n \* factorial(n – 1)

Error since implementation of factorial is not tail recursive.

In Scala, you have a recursive function that calls itself as its last action, then you can reuse the stack frame of that function.

True

The Scala function gcd is not a tail recursive function.

def gcd(a: Int, b: Int): Int =

if (b == 0) a else gcd(b, a % b)

True

The Scala compiler checks for tail recursion using \_\_\_\_\_ annotation.

@tailrec

Using Tail Recusion in Scala

Instrunctions:

* Write a function gcd to find the Greatest Common divisor of two integer numbers input1 and input2 using Tail Recursion

Code:

import java.io.\_

import java.math.\_

import java.security.\_

import java.text.\_

import java.util.\_

import java.util.concurrent.\_

import java.util.function.\_

import java.util.regex.\_

import java.util.stream.\_

import scala.collection.immutable.\_

import scala.collection.mutable.\_

import scala.collection.concurrent.\_

import scala.concurrent.\_

import scala.io.\_

import scala.math.\_

import scala.sys.\_

import scala.util.matching.\_

import scala.reflect.\_

// Import your desired libraries here.

object Result {

    /\*

     \* Complete the 'GCDImplementation' function below.

     \*

     \* The function accepts following parameters:

     \*  1. INTEGER input1

     \*  2. INTEGER input2

     \*/

    def GCDImplementation(input1: Int, input2: Int) {

        // Put your code here to implement tail recursion.

        def gcd(a:Int, b: Int): Int = {

            if (b == 0) a else gcd(b, a % b)

            }

        // Print the result here

        println(gcd(input1, input2))

    }

}

object Solution {

    def main(args: Array[String]) {

        val input1 = StdIn.readLine.trim.toInt

        val input2 = StdIn.readLine.trim.toInt

        Result.GCDImplementation(input1, input2)

    }

}

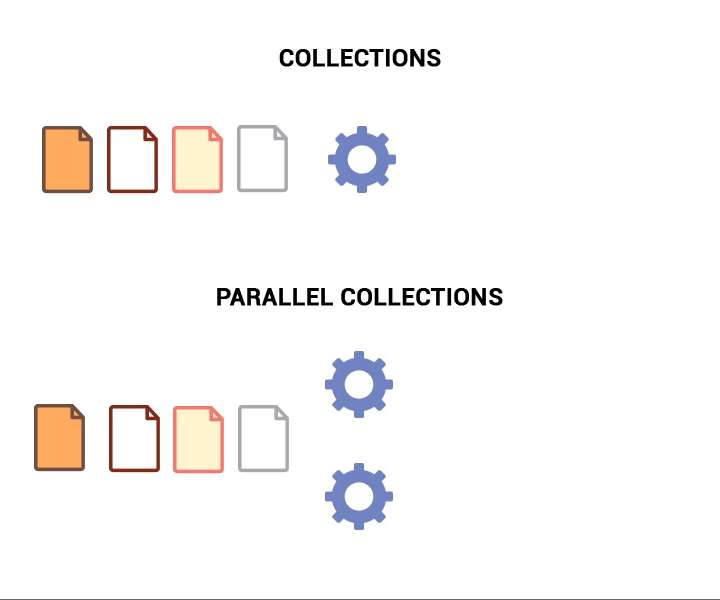
Input (stdin)

* **12**
* **18**

Your Output (stdout)

* **6**

##### Parallel Collections



The basic course on Scala covers the collection classes and the various operations available. All the methods of a collection process its elements **sequentially**. Scala also has got parallel collections which process the elements in **parallel**.

You can convert any existing sequential collection to a parallel collection using the **par** method.

##### You can convert any existing sequential collection to a parallel collection using the \_\_\_\_ method.

##### Par

You can use the new keyword and a proper import statement to create a Parallel Vector.

import scala.collection.parallel.immutable.ParVector

val pv = new ParVector[Int]

##### True

##### reating Parallel Collection

A regular Range collection

is as follows:

`val r = 1 to 10

`

You can create the parallel collection (ParRange) as shown here:

val p = r.par

In this sample, the println method will be called for 10 elements, one after the other.

r.foreach(println)

Whereas with the parallel collection, the println method will be called for the elements in parallel.

p.foreach(println)

Every parallel collection can be converted to its sequential variant using the **seq** method.

**Code:**

**Instructions:**

* Define a method uniformLists which accepts a integer parameter and return a list of length same as parameter and
* All the list items must contain the string “TCS”

object Result {

    /\*

     \* Complete the 'uniformLists' function below.

     \*

     \* The function accepts INTEGER input1 as parameter.

     \*/

    def uniformLists(input1: Int) {

        println(List.fill(input1)("TCS"))

    }

}

object Solution {

    def main(args: Array[String]) {

        val input1 = StdIn.readLine.trim.toInt

        Result.uniformLists(input1)

    }

}

Input (stdin)

* **5**

Your Output (stdout)

* **List(TCS, TCS, TCS, TCS, TCS)**