[Scala Constructs](https://play.fresco.me/course/176)

Introduction to Scala Constructs

***Welcome to Functional programming and Scala***. In this course, you will learn:

* Key concepts of Functional Programming
* Scala and how to install Scala
* Language Constructs (Variables, Collections, Higher order functions, and Function Literals - Closure)
* OOPS (Classes, Objects, Case Classes and Pattern Matching)

Concepts of Functional Programming

[**Functional Programming**](https://en.wikipedia.org/wiki/Functional_programming)**(FP) is a programming paradigm**.

It is about the style of how you develop the blocks of your program. It focuses not on how to compute, but what to compute. Functions are first class citizens in an FP.

Let us learn a few key concepts that are commonly adopted in any functional programming language viz.,

* Pure functions
* No side effects
* Immutability
* Referential transparency
* Higher-order functions

Side Effects

You must have written or will be writing lot of functions in your code. Have you realized doing any of the following actions in a function will result in **side effects**?

* Changing the value of a variable
* Updating the data structure already in place
* Setting a field in an object
* Throwing exception
* Waiting due to an error
* Printing output/content to console
* Dealing with files (read/write) and so on.

Functions that are written without any ***side effects*** are called **Pure Functions**.

##### Immutability

**If you cannot change something, that is immutable**.

An immutable object (unchangeable object) is an object whose value or state cannot be modified once created.

Immutability is not about restricting you to do something. It is more about dealing it differently.

Let us see a quick example.

val number: Int = 20

number = 30

number = 30 gives a compilation error in Scala (val is the keyword in Scala to define an immutable object)

##### Referential Transparency and Purity

**A function is supposed to be referentially transparent, if for the same input it always produces the same output.**

Can you think of an example which is against this? Let us look at the following pseudo code (too early to write Scala code!)

function getTimeAdded(input: Long) = return (System.currentTime + input)

The output of the above function will not be same even if you call with the same argument, "at different times." So the above function is not referentially transparent.

##### Why Scala

**Scala** is

* both **functional and object-oriented**.
* a modern multi-paradigm programming language.
* concise, elegant, and type-safe.

##### Popular Functional Programming languages

There are pure and impure functional languages. The list of impure languages is huge. They are impure because they allow a different style of writing.

For example, ***Scala is object-oriented, not just functional.***

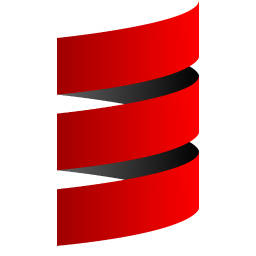
**Pure Languages:**

* Haskell
* Mercury
* AGDA

**Impure Languages:**

* Python
* Scala
* C#
* R
* F#
* Erlang

Environment for Scala Programming



You can select one of the three ways to get started with Scala!

* Download [Scala](https://www.scala-lang.org/download/)
* Use REPL and PlayGrounds commonly available over the internet for learning.
* Use SBT (Scala Build Tool) or Maven to build real projects.

Also, for Scala, you can make use of popular IDEs such as IntelliJ and Eclipse IDE. IntelliJ has good plugins for auto-completion, formatting, etc.

This course is providing embedded REPL for your hands on!

**Install Scala in Windows machine:**

**Hands on**

Scala Constructs – Handson 1

Instructions:

* This is a simple Scala program where you are supposed to print”Hello World”
* Use an object class with name HelloWorld
* Use a main method within this created object class and print the required message.

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.\_

object HelloWorld {

    def main() {

        println("Hello World")

    }

}

// Put your code here

object Solution {

    def main(args: Array[String]) {

        HelloWorld.main()

    }

}

##### Declaring and Using Variables

Let us go straight to a code snippet to understand some basic elements of the language.

Read the comments carefully in the following code. Please note:

// is to comment a line of code and /\* \*/ can be used to comment a block.

object Main extends App {

println("Hello Scala")

// initializing an immutable object/variable using val

val a: Int = 10

println(a)

// Re assignment to a val?? uncomment and see the compilation error.

// a = 20

// initializing a mutable variable using var

// a legal expression even though we haven't explicitly stated the data type of `b`.

var b = 20

// Scala compiler can infer the data type as Int.

b = 30 // we can mutate the value of `b` since its declared as `var`

}

Here object Main inherits the main method of App. App is called as trait, which is equivalent to interface in Java.

##### Declaring and Using Variables Contd...

To summarize,

* Declaring a variable with **var**, makes it mutable
* Declaring a variable with **val** makes it immutable
* Reassignment of an immutable variable is not possible
* Data type declaration happens after the variable name and a colon :
* Data type declaration is not mandatory
* In the absence of explicit data type declaration, Scala can infer the variable type based on the value assigned
* line of code or block of code can be commented using // or /\* \*/

##### Higher Order Functions

Just like we pass variables or objects as arguments to functions, we can pass around **function itself as an argument to another function** in Scala.

Even the **return type** of a function can be another function! These type of functions are called as **Higher Order Functions**.

Function Definition Explained

Scala syntax is flexible, and there are different ways you can write a function from a syntax point of view.

Let us look at a simple function definition first.

def addOne(arg: Int): Int = { arg + 1 }

* **def** is the keyword to define a method or function.
* **addOne** is a function that takes an integer as argument and returns another integer as result.

Scala notation to express it: Int => Int (You can say Int gives Int !)

**Defining Functions :**

**Instructions:**

* Define a function ‘add ’ which returns the sum of two integer numbers.
* Use an object class with name Result.
* The function ‘add’ accepts two integer parameters.

Code :

    /\*

     \* Complete the 'add' function below.

     \*

     \* The function is expected to return an INTEGER.

     \* The function accepts following parameters:

     \*  1. INTEGER op1

     \*  2. INTEGER op2

     \*/

    // Put your code here

object Result {

    def add(a: Int, b: Int): Int ={

        return a + b

    }

}

object Solution {

    def main(args: Array[String]) {

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

        val op1 = StdIn.readLine.trim.toInt

        val op2 = StdIn.readLine.trim.toInt

        val result = Result.add(op1, op2)

        printWriter.println(result)

        printWriter.close()

    }

}

Create a Higher Order function

Let us consider the following code snippet, **myHigherOrderFunction** function that takes two parameters.

* **First parameter** or **argument** is a **function** (i.e. *a function itself as an argument*). The function which is passed, when called, should be of type Int => Int. You could relate this to a function definition in the previous section.
* **Second parameter** is a value, a regular kind of parameter that we see.

def myHigherOrderFunction(argFn: Int => Int, argVal: Int): Int = {

println("Inside myHigherOrderFunction ")

println(s"\n Applying the arg function to argVal = $argVal")

argFn(argVal) // compute the argFn and return the result

}

**Another takeaway**: Look at the second print statement in the snippet and understand how to substitute a variable value in a string.

##### Calling a Higher Order Function

val myNumber = 10

val result = myHigherOrderFunction(addOne, myNumber)

println(result)

The output of the above code would be 11. Do you know how that is done?

**Important code on HigherOrder Functions**

object Result {

def addOne(a: Int): Int = {

return a + 1;

}

def myHigherOrderFunction(addOne: Int => Int, argVal: Int): Int = {

println("Inside myHigherOrderFunction")

addOne(argVal)

}

}

object Solution {

def main(args: Array[String]){

val myNumber = 10

val result = Result.myHigherOrderFunction(Result.addOne, myNumber)

println(result)

}

}

OR

**Actual approach**

object Result {

def myHigherOrderFunction(addOne: Int => Int, argVal: Int): Int = {

println("Inside myHigherOrderFunction")

addOne(argVal)

}

}

object Solution {

def main(args: Array[String]){

def addOne(a: Int): Int = {

return a + 1;

}

val myNumber = 10

val result = Result.myHigherOrderFunction(addOne, myNumber)

println(result)

}

}

**Function Literals :**

**Instructions:**

* Define a function literal named ‘multiply’ which returns the product of two integer numbers.
* Use an object with name Result to define the function literal.
* The function ‘multiply’ accepts two integer paramerters.

**Code :**

// Put your code here

    /\*

     \* Complete the 'multiply' function literal below.

     \*

     \*/

object Result {

    var multiply = (a:Int, b:Int) => a\*b// Define your function literal here

}

object Solution {

    def main(args: Array[String]) {

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

        val op1 = StdIn.readLine.trim.toInt

        val op2 = StdIn.readLine.trim.toInt

        val result = Result.multiply(op1, op2)

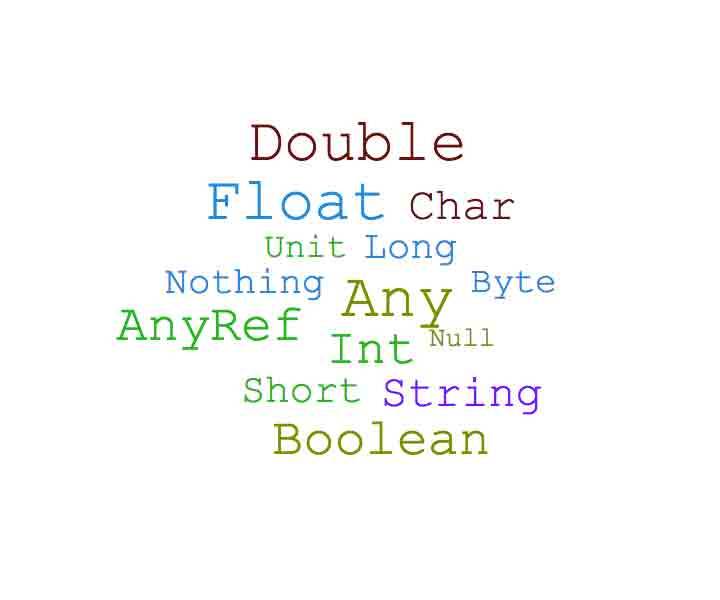
        printWriter.println(result)

        printWriter.close()

    }

}

##### Data Types in Scala



##### Function Literals

* In Scala, functions can be expressed in **function literal**
* For example: (x: Int) => x + 1 is a function literal
* Those functions can be represented by objects, which are called **function values**.

The above function can be read as, a function which takes an integer argument and returns an integer result.

You can assign it to an object also as val f = (x: Int) => x + 1

##### Using Function Literal

Here is an example snippet to show to use function literals or function values:

def myHigherOrderFunction(argFn: Int => Int, argVal: Int): Int = {

argFn(argVal) // compute the argFn and return the result

}

If the literal is not assigned to an object, it can be used as

val result = myHigherOrderFunction((a: Int) => { a + 1 }, 12).

If the same is assigned to an object f, it can be used as

val result = myHigherOrderFunction(f, 12).

Both will give the same result.

**Function literals are also known as anonymous functions**.

Important Questions:

Which is the default class used to import a list in Scala?

Ans

scala.collection.immutable

For the given statement,

val s = List(1,2,3)

What would be the type of s?

Ans : List[Int]

Is this a valid expression?

val f: Int = (x: Int) => x + 1

No

What is the preferred qualifier for defining a variable in Scala?

Val

What would be the output of following snippet?

val s = List(1,2,3,4)

val t = List(5,6,7,8)

val n = s ++ t

println(n)

List(1, 2, 3, 4, 5, 6, 7, 8)

##### Scala Collections

* List
* Filter
* forEach
* Map

**Collections** let you organize large numbers of objects.

* Scala has a rich collection library
* In simple cases, you can throw a few objects into a set or a list and not think much about it.
* For trickier cases, Scala provides a general library with several collection types, such as sequences, sets and maps.
* Each collection type comes in two variants—**mutable** and **immutable**.

Most kinds of **collections have several different implementations** that have different tradeoffs of speed, space, and the requirements on their input data.

##### Lists

Let us get started with List, one of the most used collections in Scala.

**List**, as the name goes, is similar to arrays. It is a **collection of elements of the same type**.

Here is an example to create an integer list:

val intList: List[Int] = List(1,2,3).

As Scala is a language with lot of syntactic sugar and built-in type inference features, programmers can write the above expression as

val intList = List(1,2,3) // this is valid too.

##### Map

Collections in Scala has a lot of built-in functions, and many of them are of type higher order functions. Let us see an example.

// function to convert an integer to a string

def covertToString(arg: Int): String = arg.toString

Now, the following snippet would convert a list Integer values to a list of String, using map function.

val newList: List[String] = intList.map(covertToString)

// convertToString is a function which is passed onto map function in list.

The function map in List type is a higher order function. It accepts a function that operates on one element of a List at a time. The argument function maps one element of list to another type say, U. i.e once you apply the map function on a list of type List[T], the result you get back is a List[U].

**Note: U can be same as T as well**

Using user-defined function

object GfG

{

    // square of an integer

    def square(a:Int):Int

    =

    {

        a\*a

    }

    // Main method

    def main(args:Array[String])

    {

        // source collection

        val collection = List(1, 3, 2, 5, 4, 7, 6)

        // transformed collection

        val new\_collection = collection.map(square)

        println(new\_collection)

    }

}

Using anonymous function

object GfG

{

    // Main method

    def main(args:Array[String])

    {

        // source collection

        val collection = List(1, 3, 2, 5, 4, 7, 6)

        // transformed collection

        val new\_collection = collection.map(x => x \* x )

        println(new\_collection)

    }

}

**Code:**

**List Functions**

In this Hands-on, Use the list List(1,2,3,4,5,6) as a sample list to perform various list operations. Below are the instructions to be followed.

**Instructions :**

* Write a code to print head of the list, tail of the list, length of the list, and reverse of the list.
* Use an object class with name Result.
* Define a function with name ListFunctions inside the object class

**Code:**

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.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.\_

// Put your code here

object Result{

    def ListFunctions() = {

        val sampleList: List[Int] =  List(1,2,3,4,5,6)

        println(sampleList.head)

        println(sampleList.tail)

        println(sampleList.length)

        println(sampleList.reverse)

    }

}

object Solution {

    def main(args: Array[String]) {

        Result.ListFunctions()

    }

}

**Output:**

* **1**
* **List(2, 3, 4, 5, 6)**
* **6**
* **List(6, 5, 4, 3, 2, 1)**

**String length – str.length()**

**Collection length – coll.length**

##### Collection Hierarchy

Scala collections orderly differentiate immutable and mutable collections.

* A **mutable collection** can be extended or updated in place. i.e. user can include, change or exclude elements of a collection as a side effect.
* By contrast, **immutable collections**, never change. Still, a user has operations that simulate updates, removals, or additions. However, these operations will return a new collection in each case and allow the old collection unchanged, as well.

Collection Classes

* The entire lot of **collection classes** are available in the scala.collection package or one of the sub-packages of scala.collection - generic, immutable, and mutable.
* Most [collection classes](http://docs.scala-lang.org/overviews/collections/overview.html) needed by client code exist in three variants, which are located in packages scala.collection, scala.collection.immutable, and scala.collection.mutable, respectively.
* Each variant features distinct characteristics on mutability.

Note: ***By default, Scala always takes collection from the immutable hierarchy.***

For instance,

* If you just write Set without any prefix, or without having imported Set from somewhere, you will get an immutable set.
* If you write Iterable, you will get an immutable iterable collection, because these are the default bindings imported from the Scala package.
* To have the mutable default versions, you have to write explicitly collection.mutable.Set, or collection.mutable.Iterable

##### Filtering

There are plenty of built-in functions provided in Scala for all collections. We will look at a few of the commonly used ones.

Here is an example of filter and \_some more syntactic sugars in Scala. It is fun. Read the commented part carefully.

object Main extends App {

val lst = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

// example for filtering all elements greater than 5

val g5 = lst.filter(i => i > 5)

println(g5)

}

* The above statement also shows usage of an anonymous function. ie i => i > 5 is actually of type Int => Boolean. type of i is not explicitly given as Scala understands that it is an Integer.
* Writing it as i: Int => i > 5 is also correct.
* Another simplification in terms of Syntactic sugar is to write the expression as val g5 = lst.filter(\_ > 5). **\_** (underscore) will be replaced by the elements of the list by Scala compiler for you.

##### foreach

foreach is an example of a method that can have side effects. foreach does not return anything. It can be used for writing the output to disk, database, printing, etc.

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

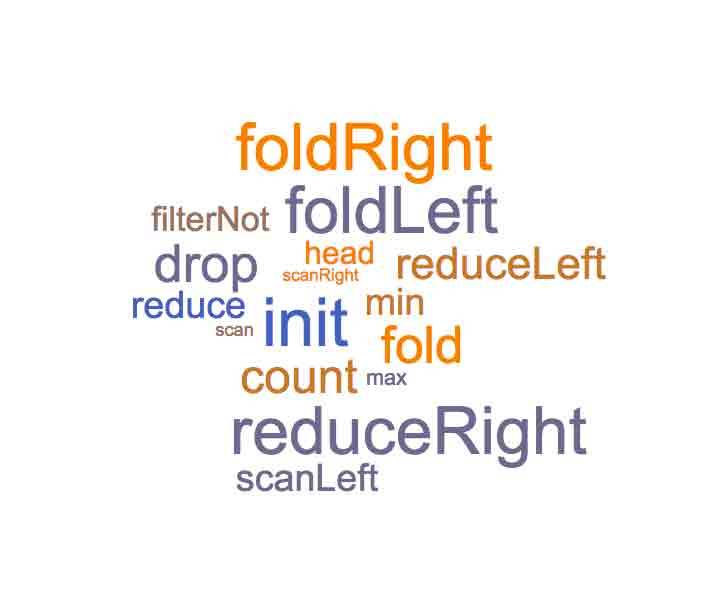
intList.foreach(println) // foreach is of return type `Unit`

Above snippet will print the result to stdout.

**Unit is the return type to mean there is nothing returned**. Hence the methods of return type Unit can have side effects, i.e., they are not pure functions.

Does it make sense with the functional programming concepts we learned?

##### Other Important Functions in Collections



Refer the [scala library](http://www.scala-lang.org/api/2.12.0/scala/collection/TraversableLike.html)

Do some exercises on your own. We will cover few of the complex ones in the upcoming courses.

**Code:**

**Using Filters in Scala**

**Instructions:**

* ListFilter takes integer input as a function parameter.
* A list must be generated with the input number of elements in the list starting from zero.
* On the first line, Print the list of integers.
* On the second line, Print the even numbers from the list using the filter function.

**Code:**

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.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 'ListFilter' function below.

     \*

     \* The function accepts INTEGER var1 as parameter.

     \*/

    def ListFilter(var1: Int) {

       // Put your code here

       val intList: List[Int] =  List.range(0, var1)

       println(intList)

       val evenNum = intList.filter(i => i % 2 == 0) // or intList.filter(\_%2 ==0)

       println(evenNum)

    }

}

object Solution {

    def main(args: Array[String]) {

        val var1 = StdIn.readLine.trim.toInt

        Result.ListFilter(var1)

    }

}

**Output :**

* **List(0, 1, 2, 3, 4)**
* **List(0, 2, 4)**

What is flatMap?

We have already tried some examples with map. **flatMap** is very similar to map. However, there are some differences concerning the functions that are passed onto them. Both map and flatMap can be applied on many other data structures and where they are supported.

Let us use List as an example here to understand about flatMap easily.

To make it clear,

* map accepts a function that returns U for a given T. Here, T => U, hope you remember the previous example about Int => Int. T and U are generic types here.
* flatMap accepts a function that returns a List[U] for every T. ie the function passed onto it should be of type T => List[U]. The results are finally flattened to create a single List[U].

It might be looking slightly complicated. Some examples would make it clear.

##### Example for flatMap

Following snippet gives an example for flatMap.

object Main extends App {

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

def returnTwo(arg: Int): List[Int] = List(arg, arg)

val newList = intList.flatMap(returnTwo)

println(newList)

}

output of the above snippet would be List(1,1,2,2,3,3,4,4,5,5)

Hope that is clear now!

**Code :**

**Using FlatMaps in Scala**

**Instructions:**

* flatMapUsage takes integer input as function parameter.
* A list numberList must be generated with the input number of elements in the list starting from 1.
* Each number in the List generated numberList must be used to create separate lists with elements equal to the number starting from 1 and named as listGenerator.
* Next join all the elements in the lists generated in the previous step using flatmap and name it as resultList.
* On the first line, print the combined list of elements.
* On the second line, Print the length of the list generated.

Code:

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.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 'flatMapUsage' function below.

     \*

     \* The function accepts INTEGER var1 as parameter.

     \*/

    def flatMapUsage(var1: Int) {

         // Put your code h

         val listNum: List[Int] = List.range(1, var1+ 1)

         def listGenerator(arg: Int): List[Int] = List.range(1, arg+1)

         val resultList = listNum.flatMap(listGenerator)

         println(resultList)

         println(resultList.length)

    }

}

object Solution {

    def main(args: Array[String]) {

        val var1 = StdIn.readLine.trim.toInt

        Result.flatMapUsage(var1)

    }

}

Output:

* **List(1, 1, 2, 1, 2, 3)**
* **6**

##### Objects

Scala is a **pure object oriented language** and it extends the use of classes, objects, interfaces, etc. to programmers.

While working on building applications using Scala, you can decompose your program into classes, objects, etc. to deal with complexities.

In other words, a class should be responsible for **a reasonable amount of functionality**.

In the process of designing your classes, you can also design interfaces to those classes that abstract away the details of their implementation.

Objects and Variables

When writing the code of a Scala program, you create and interact with objects. To interact with a particular object, you can use a variable that refers to the object.

You can define a variable with either a val or var, and assign an object to it with =.

For example,

* When you write **val i = 1**, you create an *Int* object with the value 1 and assign it to a variable. In this case, a val named i.
* Similarly, when you write **var s = "Happy"**, you create a *String* object with the value "Happy" and assign it to a variable, a var named as s.

Singleton Objects

One way in which Scala is more object-oriented than Java is that classes in Scala cannot have static members.

Instead, Scala has singleton objects. A singleton object definition looks like a class definition, except instead of the keyword class *you use the keyword object*. Here’s an example:

*Don't worry about the logic in the following snippet. However you may explore what is for comprehension in Scala.*

object simpleObjectExample {

val defaultValue = 20

def doSomething(arg: Int) = { arg \* 2 \* 3 }

}

You can use the functions or members as objectName.member.

for ex: simpleObjectExample.doSomething(100)

We have already used singleton objects in our previous sections, in various places. Do you remember?

##### Classes

A class is considered as a blueprint for objects. Once you define a class, you can create objects of it using the keyword new. For example, given the class definition:

class Home {

// class definition goes here

// define your members here (both variables and functions)

}

You can create Home objects (also known as instances of class Home) with:

new Home

A user can assign the object to a variable, for later reference and usage.

val h1 = new Home

you can construct many objects from one class:

val h2 = new Home

val h3 = new Home

You now have three Home objects.

##### Case Class

case classes are similar to classes. You need to a put a keyword case in front of class.

Is that all ? No.

You can build objects of them without using new keyword.

How do I create a case class.

case class Fresco(specialisation: String, courseName: String)

Now creating an object of type Fresco is easy enough as

val obj = Fresco("modern data platform", "functional programming in Scala")

There is no *new* keyword used, no setter methods used.

***case classes can have methods just like normal classes***

##### Pattern Matching

**Pattern matching** is very similar to switch case structures that you would have seen other programming languages. However, Pattern Matching in Scala has lot more to do!

Let us start with a simple example.

**Matching on Values**

val times = 1

times match {

case 1 => "one"

case 2 => "two"

// \_ matches everything else

case \_ => "some other number"

}

You can also use the concept called guards along with pattern matching. An example is here.

// Note the if conditions given in case

times match {

case i if i == 1 => "one"

case i if i == 2 => "two"

case \_ => "some other number"

}

##### Pattern Matching on types

We have seen one case of pattern matching, that was matching on values. Now let us look at an example of matching on types, sounds interesting?

You can use a match to handle values of different types differently.

// Any is the super type of all built-in types in Scala.

In this example, argument o is matched against a type, whether it is Double or Int before doing an operation.

def bigger(o: Any): Any = {

o match {

case i: Int if i < 0 => i - 1

case i: Int => i + 1

case d: Double if d < 0.0 => d - 0.1

case d: Double => d + 0.1

case text: String => text + "s"

}

}

##### Pattern matching on case classes

case classes are designed to be used with pattern matching.

abstract class Notification

case class Email(

sender: String,

title: String,

body: String) extends Notification

case class SMS(

caller: String,

message: String) extends Notification

case class VoiceRecording(

contactName: String,

link: String) extends Notification

Notification is an abstract super class which has three concrete Notification types implemented with case classes Email, SMS, and VoiceRecording.

##### Pattern Matching on Case Classes Contd...

Now we can do pattern matching on these case classes:

def showNotification(notification: Notification): String = {

notification match {

case Email(email, title, \_) =>

s"You got an email from $email with title: $title"

case SMS(number, message) =>

s"You got an SMS from $number! Message: $message"

case VoiceRecording(name, link) =>

s"you received a Voice Recording from $name! Click the link to hear it: $link"

}

}

val someSms = SMS("12345", "Are you there?")

val someVoiceRecording = VoiceRecording("Tom", "voicerecording.org/id/123")

// prints You got an SMS from 12345! Message: Are you there?

println(showNotification(someSms))

// you received a Voice Recording from Tom! Click the link to hear it: voicerecording.org/id/123

println(showNotification(someVoiceRecording))

**code:**

**Pattern Matching in scala**

**Instructions:**

* PatternMatching takes the integer input from command line.
* Based on the input, write a code using match to print Negative/Zero is input when input is less than or equal to 0.
* Print Evn number is given when input is even, and
* Print Odd number is given when input is odd

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 'PatternMatching' function below.

     \*

     \* The function accepts INTEGER var1 as parameter.

     \*/

    def PatternMatching(var1: Int) {

        var1 match{

            case i if i <= 0 => println("Negative/Zero is input")

            case i if i % 2 == 0 => println("Even number is given")

            case i if i % 2 != 0 => println("Odd number is given")

        }

    }

}

object Solution {

    def main(args: Array[String]) {

        val var1 = StdIn.readLine.trim.toInt

        Result.PatternMatching(var1)

    }

}

##### Traits

Traits are applied to share fields and interfaces between classes. These are same as Java 8’s interfaces. Objects and classes can extend traits. However, traits cannot be instantiated and hence have no parameters.

**Defining a trait**

A minimal trait is simply the keyword trait and an identifier:

ex: trait HairColor

extends is the keyword to inherit from a trait.

##### Traits Example

Let us look at an example

for trait.

trait BaseSoundPlayer {

def play

def close

def pause

def stop

def resume

}

If a class implements one trait it will use the extends keyword:

class Mp3SoundPlayer extends BaseSoundPlayer {

def play {}

def close {}

def pause {}

def stop {}

def resume {}

}

One trait can extend another trait:

If a class extends a trait but does not implement the methods defined in that trait, it must be declared abstract:

##### Implementing Multiple Traits

If a class implements multiple traits, it will extends the first trait (or a class, or abstract class), and then use with for other traits:

Here is an example

abstract class Animal {

def speak

}

trait WaggingTail {

def startTail

def stopTail

}

trait FourLeggedAnimal {

def walk

def run

}

class Dog extends Animal with WaggingTail with FourLeggedAnimal {

// implementation code here ...

}

**Code :**

**Basic Class Implementation:**

**Instructions:**

* Define a Class Rectangle. Class Rectangle has a constructor taking two integer parameters, length and breadth.
* It has two functions area and perimeter which calculate the respective parameters for the Rectangle object created.
* Print the values of area and perimeter in the respective class methods.

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.\_

class Rectangle(length: Int, breadth: Int) {

    def area() {

        val areaOfRectangle: Int = length \* breadth

        println(areaOfRectangle)

    }

    def perimeter() {

        val perimeterOfRectangle: Int = 2\*(length + breadth)

        println(perimeterOfRectangle)

    }

}

// Put your code here

object Solution {

    def main(args: Array[String]) {

        val length = StdIn.readLine.trim.toInt

        val breadth = StdIn.readLine.trim.toInt

        // Result.Rectangle(length, breadth)

        var rect = new Rectangle(length, breadth);

        rect.area();

        rect.perimeter();

    }

}

**Code:**

**Using case classes in scala**

**Instructions:**

* Follow the instructions which are given as comments and proceed through the challenge.
* Create two book objects with the following details
* 1.
* Name : Scala for the Impatient
* Author: Cay S. Horstmann
* 2.
* Name: Scala Cookbook
* Author: Alvin Alexander
* Print the message in the following format: “Author of this {name} is {author}”

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 {

    // Create a case class book with name(string) and author(string)

    case class Book(name: String, author: String)

    def describe() {

        //Create the two case class objects as described in the statement

        //Print the details here

        val obj1 = Book("Scala for the Impatient", "Cay S. Horstmann")

        val obj2 = Book("Scala Cookbook","Alvin Alexander")

        println(s"Author of this ${obj1.name} is ${obj1.author}")

        println(s"Author of this ${obj2.name} is ${obj2.author}")

    }

}

object Solution {

    def main(args: Array[String]) {

        Result.describe()

    }

}

**Code:**

**Working on traits in Scala**

**Instructions :**

* Write trait-ArithmeticOperations such that it has the variables x, y, and functions add, subtract, multiply, and divide.
* The class Variables extend ArithmeticOperations and have a constructor taking in two integers.
* On successful execution the code must give the desired output of each operation.
* Return the results of add, subtract, multiply and divide to the respective methods.

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{

    trait ArithmeticOperations{

        // Put your code here

        val x:Int

        val y:Int

        def add: Int

        def subtract: Int

        def multiply: Int

        def divide: Int

    }

    class Variables(op1: Int, op2: Int) extends ArithmeticOperations{

        // Put your code here

        val x = op1

        val y = op2

        def add() = x + y

        def subtract() = x - y

        def multiply() =  x \* y

        def divide() = x/y

    }

}

object Solution {

    def main(args: Array[String]) {

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

        val op1 = StdIn.readLine.trim.toInt

        val op2 = StdIn.readLine.trim.toInt

        // val result = Resultvariable(op1, op2)

        var variable = new Result.Variables(op1, op2)

        // printWriter.println(result)

        printWriter.println(variable.add)

        printWriter.println(variable.subtract)

        printWriter.println(variable.multiply)

        printWriter.println(variable.divide)

        printWriter.close()

    }

}

**Code:**

**Using higher order function with lists**

**Instructions :**

* ListHigherOrder takes an integer input
* Based on the input write a code to generate a list intList, upto a given integer stating from 1. When input is 3, List(1,2,3) must be generated.
* Write a function factorial which gives the factorial of a given number using recursion
* Write a higher order function myHigherOrder Function which takes factorial function and intList, and prints the factorial of each integer in the intList.

**Code:**

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.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 {

    def Factorial(){

    }

    def ListHigherOrder(input: Int) {

     // Put your code here

        val intList: List[Int] =  List.range(1, input+1)

        def factorial(num: Int): Int = {

            def fac(x: Int, acc: Int): Int = {

                if (x == 1) acc

                else fac(x - 1, acc \* x)

            }

        fac(num, 1)

        }

        def myHigherOrderFunction(fac: (Int) => Int, intList: List[Int]) = intList.foreach((x: Int) => println(fac(x)))

        myHigherOrderFunction(factorial, intList)

    }

}

object Solution {

    def main(args: Array[String]) {

        val op1 = StdIn.readLine.trim.toInt

        Result.ListHigherOrder(op1)

    }

}

**Output :**

* **1**
* **2**
* **6**

##### Scala Constructs Course Summary

In this course, you have learnt

* Scala as functional program
* Various programming constructs
* Usage of higher order functions
* Understand Scala collections and built-in functions
* Scala is an object oriented program

Questions:

Mentioning the function return type is optional in Scala.

A True

Scala is a JVM language.

True

**Which of the following types allows only a single instance to exist in the global scope?**

1. Object

Which of the following cannot have a constructor?

1. Trait

AnyVal is the best primitive data type.

1. True

Consider the following code snippet:

def prnt = {print("scala"); 1}

def fun(a:Int,b: => Int) = print(a)

What will be the output for function call fun(prnt,5)?

scala1

Scalability is an important trait of a functional programming language.

* 1. True

What would be the output of the following code snippet?

val s = List (10, 11, 12)

val result = s.flatMap(i => List(i-1, i, i +1))

println(result)

List(9, 10, 11, 10, 11, 12, 11, 12, 13)

Object keyword is used to define a singleton class in Scala.

1. True

Scala can be run only on JVM.

1. False

Scala source code can be compiled to Java bytecode and run on a Java virtual machine (JVM). Scala can also be compiled to JavaScript to run in a browser, or directly to a native executable.

toString is a default method available for the new class definition that can be overridden.

1. True

Scala supports only single inheritance.

1. False

Consider the following code snippet:

(a: Int) => a\*a .

This code snippet is an example of \_\_\_\_\_\_\_

Anonymous function

Expressions in Scala are evaluated using a method called \_\_\_\_\_\_\_\_\_

Substitution

What would be the output of the following code snippet?

val l = List(1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 5, 5, 8)

println(l.drop(10))

List(3, 5, 5, 8)

What would be the output of the following code snippet?

val l = List(1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 5, 5, 8)

println(l.take(3))

List(1,1,1)

What would be the type inferred by Scala compiler for variable salary. val salary = 3000.00?

Double

Salary.getClass

# **Scala - Overview**

Scala, short for Scalable Language, is a hybrid functional programming language. It was created by Martin Odersky. Scala smoothly integrates the features of object-oriented and functional languages. Scala is compiled to run on the Java Virtual Machine. Many existing companies, who depend on Java for business critical applications, are turning to Scala to boost their development productivity, applications scalability and overall reliability.

Here we have presented a few points that makes Scala the first choice of application developers.

### **Scala is object-oriented**

Scala is a pure object-oriented language in the sense that every value is an object. Types and behavior of objects are described by classes and traits which will be explained in subsequent chapters.

Classes are extended by **subclassing** and a flexible **mixin-based composition** mechanism as a clean replacement for multiple inheritance.

### **Scala is functional**

Scala is also a functional language in the sense that every function is a value and every value is an object so ultimately every function is an object.

Scala provides a lightweight syntax for defining **anonymous functions**, it supports **higher-order functions**, it allows functions to be **nested**, and supports **currying**. These concepts will be explained in subsequent chapters.

### **Scala is statically typed**

Scala, unlike some of the other statically typed languages (C, Pascal, Rust, etc.), does not expect you to provide redundant type information. You don't have to specify a type in most cases, and you certainly don't have to repeat it.

### **Scala runs on the JVM**

Scala is compiled into Java Byte Code which is executed by the Java Virtual Machine (JVM). This means that Scala and Java have a common runtime platform. You can easily move from Java to Scala.

The Scala compiler compiles your Scala code into Java Byte Code, which can then be executed by the '**scala**' command. The '**scala**' command is similar to the **java** command, in that it executes your compiled Scala code.

### **Scala can Execute Java Code**

Scala enables you to use all the classes of the Java SDK and also your own custom Java classes, or your favorite Java open source projects.

### **Scala can do Concurrent & Synchronize processing**

Scala allows you to express general programming patterns in an effective way. It reduces the number of lines and helps the programmer to code in a type-safe way. It allows you to write codes in an immutable manner, which makes it easy to apply concurrency and parallelism (Synchronize).

## **Scala vs Java**

Scala has a set of features that completely differ from Java. Some of these are −

* All types are objects
* Type inference
* Nested Functions
* Functions are objects
* Domain specific language (DSL) support
* Traits
* Closures
* Concurrency support inspired by Erlang

## **Scala Web Frameworks**

Scala is being used everywhere and importantly in enterprise web applications. You can check a few of the most popular Scala web frameworks −

* [**The Lift Framework**](http://liftweb.net/)
* [**The Play framework**](http://www.playframework.org/)
* [**The Bowler framework**](https://github.com/bowler-framework/bowler-quickstart)

If you have a good understanding on Java, then it will be very easy for you to learn Scala. The biggest syntactic difference between Scala and Java is that the ';' line end character is optional.

When we consider a Scala program, it can be defined as a collection of objects that communicate via invoking each other’s methods. Let us now briefly look into what do class, object, methods and instance variables mean.

* **Object** − Objects have states and behaviors. An object is an instance of a class. Example − A dog has states - color, name, breed as well as behaviors - wagging, barking, and eating.
* **Class**− A class can be defined as a template/blueprint that describes the behaviors/states that are related to the class.
* **Methods** − A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
* **Fields** − Each object has its unique set of instance variables, which are called fields. An object's state is created by the values assigned to these fields.
* **Closure** − A **closure** is a function, whose return value depends on the value of one or more variables declared outside this function.
* **Traits** − A trait encapsulates method and field definitions, which can then be reused by mixing them into classes. Traits are used to define object types by specifying the signature of the supported methods.

## **Basic Syntax**

The following are the basic syntaxes and coding conventions in Scala programming.

* **Case Sensitivity** − Scala is case-sensitive, which means identifier **Hello** and **hello** would have different meaning in Scala.
* **Class Names** − For all class names, the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

**Example** − class MyFirstScalaClass.

* **Method Names** − All method names should start with a Lower Case letter. If multiple words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

**Example** − def myMethodName()

* **Program File Name** − Name of the program file should exactly match the object name. When saving the file you should save it using the object name (Remember Scala is case-sensitive) and append ‘**.scala**’ to the end of the name. (If the file name and the object name do not match your program will not compile).

**Example** − Assume 'HelloWorld' is the object name. Then the file should be saved as 'HelloWorld.scala'.

* **def main(args: Array[String])** − Scala program processing starts from the main() method which is a mandatory part of every Scala Program.

## **Scala Packages**

A package is a named module of code. For example, the Lift utility package is net.liftweb.util. The package declaration is the first non-comment line in the source file as follows −

package com.liftcode.stuff

Scala packages can be imported so that they can be referenced in the current compilation scope. The following statement imports the contents of the scala.xml package −

import scala.xml.\_

You can import a single class and object, for example, HashMap from the scala.collection.mutable package −

import scala.collection.mutable.HashMap

You can import more than one class or object from a single package, for example, TreeMap and TreeSet from the scala.collection.immutable package −

import scala.collection.immutable.{TreeMap, TreeSet}

# **Scala - Data Types**

|  |  |
| --- | --- |
| **Sr.No** | **Data Type & Description** |
| 1 | **Byte**  8 bit signed value. Range from -128 to 127 |
| 2 | **Short**  16 bit signed value. Range -32768 to 32767 |
| 3 | **Int**  32 bit signed value. Range -2147483648 to 2147483647 |
| 4 | **Long**  64 bit signed value. -9223372036854775808 to 9223372036854775807 |
| 5 | **Float**  32 bit IEEE 754 single-precision float |
| 6 | **Double**  64 bit IEEE 754 double-precision float |
| 7 | **Char**  16 bit unsigned Unicode character. Range from U+0000 to U+FFFF |
| 8 | **String**  A sequence of Chars |
| 9 | **Boolean**  Either the literal true or the literal false |
| 10 | **Unit**  Corresponds to no value |
| 11 | **Null**  null or empty reference |
| 12 | **Nothing**  The subtype of every other type; includes no values |
| 13 | **Any**  The supertype of any type; any object is of type *Any* |
| 14 | **AnyRef**  The supertype of any reference type |

Difference between var and val

Var – mutable variable

Val – immutable variable. Variable cannot be changed

### **Syntax**

val or val VariableName : DataType = [Initial Value]

## **Multiple assignments**

Scala supports multiple assignments. If a code block or method returns a Tuple (**Tuple** − Holds collection of Objects of different types), the Tuple can be assigned to a val variable.

### **Syntax**

val (myVar1: Int, myVar2: String) = Pair(40, "Foo")

## **Extending a Class**

You can extend a base Scala class and you can design an inherited class in the same way you do it in Java (use **extends** key word), but there are two restrictions: method overriding requires the **override** keyword, and only the **primary** constructor can pass parameters to the base constructor. Let us extend our above class and add one more class method.

Scala allows the inheritance from just one class only.

import java.io.\_

class Point(val xc: Int, val yc: Int) {

var x: Int = xc

var y: Int = yc

def move(dx: Int, dy: Int) {

x = x + dx

y = y + dy

println ("Point x location : " + x);

println ("Point y location : " + y);

}

}

class Location(override val xc: Int, override val yc: Int,

val zc :Int) extends Point(xc, yc){

var z: Int = zc

def move(dx: Int, dy: Int, dz: Int) {

x = x + dx

y = y + dy

z = z + dz

println ("Point x location : " + x);

println ("Point y location : " + y);

println ("Point z location : " + z);

}

}

object Demo {

def main(args: Array[String]) {

val loc = new Location(10, 20, 15);

// Move to a new location

loc.move(10, 10, 5);

}

}

## **Implicit Classes**

Implicit classes allow implicit conversations with class’s primary constructor when the class is in scope. Implicit class is a class marked with ‘implicit’ keyword. This feature is introduced in Scala 2.10.

**Syntax** − The following is the syntax for implicit classes. Here implicit class is always in the object scope where all method definitions are allowed because implicit class cannot be a top level class.

### **Syntax**

object <object name> {

implicit class <class name>(<Variable>: Data type) {

def <method>(): Unit =

}

}

### **Example**

Let us take an example of an implicit class named **IntTimes** with the method times(). It means the times () contain a loop transaction that will execute the given statement in number of times that we give. Let us assume the given statement is “4 times println (“Hello”)” means the println (“”Hello”) statement will execute 4 times.

The following is the program for the given example. In this example two object classes are used (Run and Demo) so that we have to save those two classes in different files with their respective names as follows.

**Run.scala** − Save the following program in Run.scala.

object Run {

implicit class IntTimes(x: Int) {

def times [A](f: =>A): Unit = {

def loop(current: Int): Unit =

if(current > 0){

f

loop(current - 1)

}

loop(x)

}

}

}

**Demo.scala** − Save the following program in Demo.scala.

import Run.\_

object Demo {

def main(args: Array[String]) {

4 times println("hello")

}

}

The following commands are used to compile and execute these two programs.

### **Command**

\>scalac Run.scala

\>scalac Demo.scala

\>scala Demo

### **Output**

Hello

Hello

Hello

Hello

**Note** −

* Implicit classes must be defined inside another class/object/trait (not in top level).
* Implicit classes may only take one non –implicit argument in their constructor.
* Implicit classes may not be any method, member or object in scope with the same name as the implicit class.

## **Singleton Objects**

Scala is more object-oriented than Java because in Scala, we cannot have static members. Instead, Scala has **singleton objects**. A singleton is a class that can have only one instance, i.e., Object. You create singleton using the keyword **object** instead of class keyword. Since you can't instantiate a singleton object, you can't pass parameters to the primary constructor. You already have seen all the examples using singleton objects where you called Scala's main method.

## **‘raw’ Interpolator**

The ‘raw’ interpolator is similar to ‘s’ interpolator except that it performs no escaping of literals within a string. The following code snippets in a table will differ the usage of ‘s’ and ‘raw’ interpolators. In outputs of ‘s’ usage ‘\n’ effects as new line and in output of ‘raw’ usage the ‘\n’ will not effect. It will print the complete string with escape letters.

|  |  |
| --- | --- |
| **‘s’ interpolator usage** | **‘raw’ interpolator usage** |
| **Program** −  object Demo {  def main(args: Array[String]) {  println(s"Result = \n a \n b")  }  } | **Program** −  object Demo {  def main(args: Array[String]) {  println(raw"Result = \n a \n b")  }  } |
| **Output** −  Result =  a  b | **Output** −  Result = \n a \n b |

## **The ‘ f ’ Interpolator**

The literal ‘f’ interpolator allows to create a formatted String, similar to printf in C language. While using ‘f’ interpolator, all variable references should be followed by the **printf** style format specifiers such as %d, %i, %f, etc.

Let us take an example of append floating point value (height = 1.9d) and String variable (name = “James”) with normal string. The following code snippet of implementing ‘f’ Interpolator. Here $name%s to print (String variable) James and $height%2.2f to print (floating point value) 1.90.

val height = 1.9d

val name = "James"

println(f"$name%s is $height%2.2f meters tall") //James is 1.90 meters tall

# **Scala - Arrays**

Scala provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type.

# For Loop in Scala

**Syntax:**

for(w <- range){

// Code..

}

Here, *w* is a variable, *<-* operator is known as a generator, according to the name this operator is used to generate individual values from the range, and the *range* is the value which holds starting and ending values. The range can be represented by using either i to j or i until j.

for(w <- 0 to 10){

println(w)

}

Prints – 0 to 10

For(w <- 0 to 10){

println(w)

}

Prints – 0 to 9

#### Multiple values in for-loop

object Main

{

    def main(args: Array[String])

    {

    // for loop with multiple ranges

        for( w <- 0 to 3; z<- 8 until 10 )

        {

            println("Value of w is :" +w);

            println("Value of y is :" +z);

        }

    }

}

**Output:**

Value of w is :0

Value of y is :8

Value of w is :0

Value of y is :9

Value of w is :1

Value of y is :8

Value of w is :1

Value of y is :9

Value of w is :2

Value of y is :8

Value of w is :2

Value of y is :9

Value of w is :3

Value of y is :8

Value of w is :3

Value of y is :9

#### Using for-loop with Collections

object Main

{

    def main(args: Array[String])

    {

        var rank = 0;

        val ranklist = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // For loop with collection

        for( rank <- ranklist){

            println("Author rank is : " +rank);

        }

    }

}

#### Using for-loop with Filters

object Main

{

    def main(args: Array[String])

    {

        val ranklist = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // For loop with filters

        for( rank <- ranklist

        if rank < 7; if rank > 2 )

        {

            println("Author rank is : " +rank);

        }

    }

}

Author rank is : 3

Author rank is : 4

Author rank is : 5

Author rank is : 6

#### Using for-loop with Yield

|  |
| --- |
| // Scala program to illustrate how to  // use for loop with yields  object Main  {      def main(args: Array[String])      {          var rank = 0;          val ranklist = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);            // For loop with yields          var output = for{ rank <- ranklist                      if rank > 4; if rank != 8 }                      yield rank            // Display result          for (rank <- output)          {              println("Author rank is : " + rank);          }      }  } |

**Output:**

Author rank is : 5

Author rank is : 6

Author rank is : 7

Author rank is : 9

Author rank is : 10

## **Scala for-loop Example using *by* keyword**

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **for**(i<-1 **to** 10 by 2){
4. println(i)
5. }
6. }
7. }

Output:

1

3

5

7

9

**While Loop**

**While (condition){**

**}**

## **Multi-Dimensional Arrays**

var myMatrix = ofDim[Int](3,3)

import Array.\_

object Demo {

def main(args: Array[String]) {

var myMatrix = ofDim[Int](3,3)

// build a matrix

for (i <- 0 to 2) {

for ( j <- 0 to 2) {

myMatrix(i)(j) = j;

}

}

// Print two dimensional array

for (i <- 0 to 2) {

for ( j <- 0 to 2) {

print(" " + myMatrix(i)(j));

}

println();

}

}

}

### **Output**

0 1 2

0 1 2

0 1 2

**Scala Collections**

// List of Strings

val fruit: List[String] = List("apples", "oranges", "pears")

// List of Integers

val nums: List[Int] = List(1, 2, 3, 4)

// Empty List.

val empty: List[Nothing] = List()

// Two dimensional list

val dim: List[List[Int]] =

List(

List(1, 0, 0),

List(0, 1, 0),

List(0, 0, 1)

)

object Demo {

def main(args: Array[String]) {

val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))

val nums = Nil

println( "Head of fruit : " + fruit.head )

println( "Tail of fruit : " + fruit.tail )

println( "Check if fruit is empty : " + fruit.isEmpty )

println( "Check if nums is empty : " + nums.isEmpty )

}

}

### **Output**

Head of fruit : apples

Tail of fruit : List(oranges, pears)

Check if fruit is empty : false

Check if nums is empty : true

object Demo {

def main(args: Array[String]) {

val fruit1 = "apples" :: ("oranges" :: ("pears" :: Nil))

val fruit2 = "mangoes" :: ("banana" :: Nil)

// use two or more lists with ::: operator

var fruit = fruit1 ::: fruit2

println( "fruit1 ::: fruit2 : " + fruit )

// use two lists with Set.:::() method

fruit = fruit1.:::(fruit2)

println( "fruit1.:::(fruit2) : " + fruit )

// pass two or more lists as arguments

fruit = List.concat(fruit1, fruit2)

println( "List.concat(fruit1, fruit2) : " + fruit )

}

}

### **Output**

fruit1 ::: fruit2 : List(apples, oranges, pears, mangoes, banana)

fruit1.:::(fruit2) : List(mangoes, banana, apples, oranges, pears)

List.concat(fruit1, fruit2) : List(apples, oranges, pears, mangoes, banana)

## **Creating Uniform Lists**

You can use **List.fill()** method creates a list consisting of zero or more copies of the same element.

object Demo {

def main(args: Array[String]) {

val fruit = List.fill(3)("apples") // Repeats apples three times.

println( "fruit : " + fruit )

val num = List.fill(10)(2) // Repeats 2, 10 times.

println( "num : " + num )

}

}

### **Output**

fruit : List(apples, apples, apples)

num : List(2, 2, 2, 2, 2, 2, 2, 2, 2, 2)

## **Reverse List Order**

List.reverse

# **Scala - Sets**

## **Concatenating Sets**

object Demo {

def main(args: Array[String]) {

val fruit1 = Set("apples", "oranges", "pears")

val fruit2 = Set("mangoes", "banana")

// use two or more sets with ++ as operator

var fruit = fruit1 ++ fruit2

println( "fruit1 ++ fruit2 : " + fruit )

// use two sets with ++ as method

fruit = fruit1.++(fruit2)

println( "fruit1.++(fruit2) : " + fruit )

}

}

object Demo {

def main(args: Array[String]) {

val fruit1 = Set("apples", "oranges", "pears")

val fruit2 = Set("mangoes", "banana")

// use two or more sets with ++ as operator

var fruit = fruit1 ++ fruit2

println( "fruit1 ++ fruit2 : " + fruit )

// use two sets with ++ as method

fruit = fruit1.++(fruit2)

println( "fruit1.++(fruit2) : " + fruit )

}

}

### **Output**

fruit1 ++ fruit2 : Set(banana, apples, mangoes, pears, oranges)

fruit1.++(fruit2) : Set(banana, apples, mangoes, pears, oranges)

## **Find Max, Min Elements in a Set**

### **Example**

object Demo {

def main(args: Array[String]) {

val num = Set(5,6,9,20,30,45)

// find min and max of the elements

println( "Min element in Set(5,6,9,20,30,45) : " + num.min )

println( "Max element in Set(5,6,9,20,30,45) : " + num.max )

}

}

## **Find Common Values Insets**

You can use either **Set.&** method or **Set.intersect** method to find out the common values between two sets.

object Demo {

def main(args: Array[String]) {

val num1 = Set(5,6,9,20,30,45)

val num2 = Set(50,60,9,20,35,55)

// find common elements between two sets

println( "num1.&(num2) : " + num1.&(num2) )

println( "num1.intersect(num2) : " + num1.intersect(num2) )

}

}

### **Output**

num1.&(num2) : Set(20, 9)

num1.intersect(num2) : Set(20, 9)

# **Scala - Maps**

Scala map is a collection of key/value pairs. Any value can be retrieved based on its key. Keys are unique in the Map, but values need not be unique.

object Demo {

def main(args: Array[String]) {

val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF", "peru" -> "#CD853F")

val nums: Map[Int, Int] = Map()

println( "Keys in colors : " + colors.keys )

println( "Values in colors : " + colors.values )

println( "Check if colors is empty : " + colors.isEmpty )

println( "Check if nums is empty : " + nums.isEmpty )

}

}

## **Concatenating Maps**

You can use either **++** operator or **Map.++()** method to concatenate two or more Maps, but while adding Maps it will remove duplicate keys.

object Demo {

def main(args: Array[String]) {

val colors1 = Map("red" -> "#FF0000", "azure" -> "#F0FFFF", "peru" -> "#CD853F")

val colors2 = Map("blue" -> "#0033FF", "yellow" -> "#FFFF00", "red" -> "#FF0000")

// use two or more Maps with ++ as operator

var colors = colors1 ++ colors2

println( "colors1 ++ colors2 : " + colors )

// use two maps with ++ as method

colors = colors1.++(colors2)

println( "colors1.++(colors2)) : " + colors )

}

}

### **Output**

colors1 ++ colors2 : Map(blue -> #0033FF, azure -> #F0FFFF,

peru -> #CD853F, yellow -> #FFFF00, red -> #FF0000)

colors1.++(colors2)) : Map(blue -> #0033FF, azure -> #F0FFFF,

peru -> #CD853F, yellow -> #FFFF00, red -> #FF0000)

## **Print Keys and Values from a Map**

object Demo {

def main(args: Array[String]) {

val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF","peru" -> "#CD853F")

colors.keys.foreach{ i =>

print( "Key = " + i )

println(" Value = " + colors(i) )}

}

}

### **Output**

Key = red Value = #FF0000

Key = azure Value = #F0FFFF

Key = peru Value = #CD853F

## **Check for a key in Map**

You can use either **Map.contains** method to test if a given key exists in the map or not. Try the Following example program to key checking.

object Demo {

def main(args: Array[String]) {

val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF", "peru" -> "#CD853F")

if( colors.contains( "red" )) {

println("Red key exists with value :" + colors("red"))

} else {

println("Red key does not exist")

}

if( colors.contains( "maroon" )) {

println("Maroon key exists with value :" + colors("maroon"))

} else {

println("Maroon key does not exist")

}

}

}

### **Output**

Red key exists with value :#FF0000

Maroon key does not exist

# **Scala - Tuples**

Scala tuple combines a fixed number of items together so that they can be passed around as a whole. Unlike an array or list, a tuple can hold objects with different types but they are also immutable.

To access elements of a tuple t, you can use method t.\_1 to access the first element, t.\_2 to access the second, and so on. For example, the following expression computes the sum of all elements of t.

object Demo {

def main(args: Array[String]) {

val t = (4,3,2,1)

val sum = t.\_1 + t.\_2 + t.\_3 + t.\_4

println( "Sum of elements: " + sum )

}

}

### **Output**

Sum of elements: 10

## **Iterate over the Tuple**

object Demo {

def main(args: Array[String]) {

val t = (4,3,2,1)

t.productIterator.foreach{ i =>println("Value = " + i )}

}

}

### **Output**

Value = 4

Value = 3

Value = 2

Value = 1

## **Swap the Elements**

t.swap

# **Scala - Options**

Scala Option[ T ] is a container for zero or one element of a given type. An Option[T] can be either **Some[T]** or **None** object, which represents a missing value. For instance, the get method of Scala's Map produces Some(value) if a value corresponding to a given key has been found, or **None** if the given key is not defined in the Map.

def findPerson(key: Int): Option[Person]

object Demo {

def main(args: Array[String]) {

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

println("capitals.get( \"France\" ) : " + capitals.get( "France" ))

println("capitals.get( \"India\" ) : " + capitals.get( "India" ))

}

}

### **Output**

capitals.get( "France" ) : Some(Paris)

capitals.get( "India" ) : None

object Demo {

def main(args: Array[String]) {

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

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

println("show(capitals.get( \"India\")) : " + show(capitals.get( "India")) )

}

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

case Some(s) => s

case None => "?"

}

}

### **Output**

show(capitals.get( "Japan")) : Tokyo

show(capitals.get( "India")) : ?

## **Using getOrElse() Method**

object Demo {

def main(args: Array[String]) {

val a:Option[Int] = Some(5)

val b:Option[Int] = None

println("a.getOrElse(0): " + a.getOrElse(0) )

println("b.getOrElse(10): " + b.getOrElse(10) )

}

}

### **Output**

a.getOrElse(0): 5

b.getOrElse(10): 10

## **Using isEmpty() Method**

object Demo {

def main(args: Array[String]) {

val a:Option[Int] = Some(5)

val b:Option[Int] = None

println("a.isEmpty: " + a.isEmpty )

println("b.isEmpty: " + b.isEmpty )

}

}

a.isEmpty: false

b.isEmpty: true

# **Scala - Iterators**

An iterator is not a collection, but rather a way to access the elements of a collection one by one. The two basic operations on an **iterator it** are **next** and **hasNext**. A call to **it.next()** will return the next element of the iterator and advance the state of the iterator. You can find out whether there are more elements to return using Iterator's **it.hasNext** method.

object Demo {

def main(args: Array[String]) {

val it = Iterator("a", "number", "of", "words")

while (it.hasNext){

println(it.next())

}

}

}

### **Output**

a

number

of

words

## **Find Min & Max Valued Element**

object Demo {

def main(args: Array[String]) {

val ita = Iterator(20,40,2,50,69, 90)

val itb = Iterator(20,40,2,50,69, 90)

println("Maximum valued element " + ita.max )

println("Minimum valued element " + itb.min )

}

}

## **Find the Length of the Iterator**

object Demo {

def main(args: Array[String]) {

val ita = Iterator(20,40,2,50,69, 90)

val itb = Iterator(20,40,2,50,69, 90)

println("Value of ita.size : " + ita.size )

println("Value of itb.length : " + itb.length )

}

}

# **Scala - Traits**

A trait encapsulates method and field definitions, which can then be reused by mixing them into classes.

trait Equal {

def isEqual(x: Any): Boolean

def isNotEqual(x: Any): Boolean = !isEqual(x)

}

class Point(xc: Int, yc: Int) extends Equal {

var x: Int = xc

var y: Int = yc

def isEqual(obj: Any) = obj.isInstanceOf[Point] && obj.asInstanceOf[Point].x == y

}

object Demo {

def main(args: Array[String]) {

val p1 = new Point(2, 3)

val p2 = new Point(2, 4)

val p3 = new Point(3, 3)

println(p1.isNotEqual(p2))

println(p1.isNotEqual(p3))

println(p1.isNotEqual(2))

}

}