**Scala - Getting Started**

**What is Scala?**

General Purpose programming language

Support bothe Object Oriented and Functional Programming

Multi-paradigm:functional, imperative,Object Oriented

Open Source

Compiler based

The major advantage of Scala is the **JVM** (**Java Virtual Machine**).

Created by Martin Odersky (2001 -> released on 2003)

Typesafe reactive platform was supported in 2011

(Provide reassurance to enterprise companies to use Scala)

Scala binds the Datatype to the variable in its entire **scope** - Statically typed

**Why Scala for Data Science?**

Capable to work with the data which is stored in a **Distributed** fashion

A scalable Language.

supports multiple type **Constructs**

REPL Interactive utility

-Read

-Evaluate

-Print

-Loop

(Immediate feedback given)

**Variables**

* Mutable - val
* Immutable - var

**Variable Declaration**

Keywords: var/ val

Ex: val v1: String = “hello world” O/P - String

With Inference :

Ex: val v1 = “hello world” O/P - String

- In Scala these inferences inherited from **Scala.Any** (Every value in

scala is considered as a kind of object)

Scala.Any

* Scala.AnyVal
* Scala.AnyRef

Variable Declaration: even more simpler

Ex: “hello world” \*This will create a new String variable and assign

String value to it.

In Scala code: Semicolon is not necessary. White Space will

terminate a command.

**Scala methods:**

Def methodName ( params : dataType ) : ReturnType = method body

**Scala Data Types: (Scalar)**

Integer (Int)

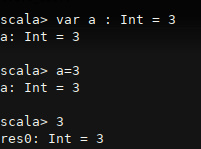
Long (Long or l )

Character (Char)

Double

Float

Variable creation ways:

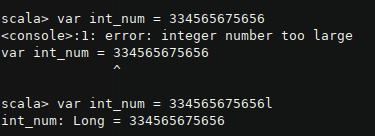


Note: Creating variables without data types:

Numbers will be considered as Integers. If they are too long or contains fractions, will give an error when assigning without Data type.

To overcome:

Make it Long type by adding **l** to the end.



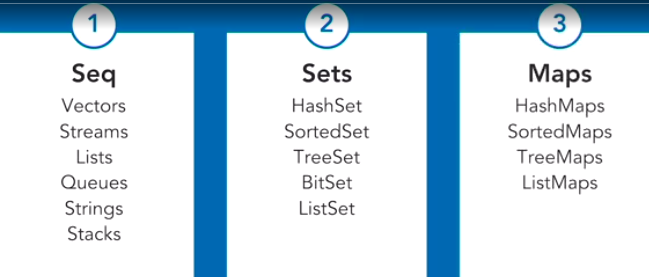
Same theory applies to Double and float, initially any value with a fraction would be double, specify **f** at the end to make them float.

**Scala Collections**

Sequences

Sets

Maps



Arrays:

Val a = Array(val1,val2,val3)

Call array element by index: (0 based) / assigning values

a(1) → val2

Array.length → Int: length

Note:

Arrays are objects in scala. Can be created using **new** keyword.



If elements are not defined at intialtion, default values for relevant data type will be stored(ex: Integer → 0)

**Creating Multi-dimensional Arrays:**

Array.ofDim[data\_type](x,y)

Ex: **Array.ofDim[Int](10,10)**

Will result in a 10x10 array with 0 s.

Array functions package can be imported as

**Import Array.\_**

After importing, we can use these methods on created arrays.

Ex: **concat(arr1, arr2)**

To see all the available methods for a collection,

**col\_name. (and press tab)**

Ex: arr1. (press Tab)

**Vectors:**

* Is a immutable collection.
* Elements should be in same data type like arrays.

Creating a Vector



Range:

* Data Structure for representing Integer values
* Immutable collection
* Creates value between a given range

Creating a range:

Val range1 = 1 to 12



Using **new** keyword:



**Val range2: Range = new Range(tart, end, step value)**

**Scala Maps**

* Groups of key: value pairs



* Immutable
* Key and Value both should not required to be in same data type.

Get list of Keys/values:

**Map1.keys map1.values**

Get value of a specific key:

**Map1 get “key1”**

Will return the value of the given key.

* If value is not found, “none” is returned.

Method 2 : **Map1(“key”)**

Check a key is contains

**Map1 contains “Mother”**  o/p true/false

Give a meaningful output if no values are found:

**Map1 getOrElse(“Key”, “Value Not Found”)**

**Scala Expressions:**

Arithmetic Expressions: As usual in Java

Logical Expressions: ex: (3>4) && (5>2)

Ex: !(3>4)

Print to console: same as Java

println({

| a = 2\*3

| a+5

}) o/p : 11

**Scala Functions**

Def funcName(arg1: Dtype , arg2: Dtype) : Dtype = { func body }

**Note: Unit in Scala**

The **Unit** type in **Scala** is used as a return statement for a function when no value is to be returned. **Unit** type can be e compared to void data type of other programming languages like Java.

**Scala Objects**

* Scala is an OOP Language
* Classes and Objects are present

Collections (arrays ,map etc) are objects, Primitive datatypes can be also invoked as objects. So then their methods can be called.

Ex: array.sorted() [if string: sorted alphabetically]

**Defining a class**

Class className (

Var var1: Dtype:,

Val v2: Dtype;

…. )

**Initiation an object:**

Val obj1 = new className( val1, val2 )

**Accessing class variables and methods:**

Same as Java

Ex: obj1.var1; obj1..v2;

For a public class,

**object. (press tab)**

Will show all possibly accessible variables and methods

Private class variables:

Ex: **Class className ( private va1: Int );**

\*Private members are not listed in the above Tab view

Class methods:

Class className( public val1: 50,

def move(a: Int, b: Int) : Unit {

println( a+b ) } )

***\*Note: Scala functions are not required to indicate a return value necessarily.***

**Parallel Processing with Scala:**

**Parallel Collections**

* ParArrays
* ParVectors
* ParHashMap

2 ways to create parallel collections:

Create new

Convert existing sequential collection into parallel

Ex: create rang r1

**r1.par**

(type: scala.collection.parrerel.immutable.parRange)

**Mapping functions over Parallel Collections**

Ex: val v = (1 to 100).toArray()

* Create a range and convert it to an array.

Array.par → convert array to ParArray

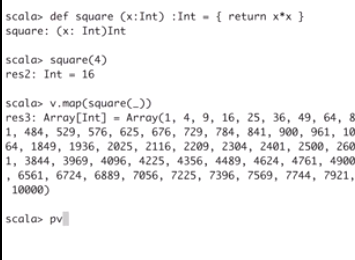
In Scala , represent all elements values inside a collection:  **\_ (underscore)**

Ex: parArray.map(\_ \* 2) : multiply each value by 2

Difference between Parallel and sequential:

* Sam results, but parallel way is more faster

**Note: applying function as a map:**



**filter() method**

Pv = array 0f 1 to 1000 range

Pv.filter(\_ > 500)

o/p ( 501, 502, …., 1000 )

**filterNot()**  ← Opposite of the filter() method

**When and When not to use parallel collections**

* Converting sequence to parallel is expensive sometimes (need to copy all elements)
* So if the dataset is too large (at least 1000+) initiate it parallely at the first place.
* If the **order of operations matters** not advised to use parallel collections

**PostgreSQL Introduction**

Create a Database: createdb database\_name

Create a new user: createuser

**Note: setting classpath in Scala:**

*Scala -classpath path\_to\_jar\_file (ex: JDBC)*

After connecting to SQL,

We can run below commands:

**Executing a Query:**

**Val resultSet = con.executeQuery(“select \* from users”);**

*Resultset ( DataType) - java.sql.ResultSet*

To get a value from the resultSet:

**ResultSet.getString(“col\_name”);**

**ResultSet.getInteger();**

Working with prepared statements:

* Prepared statements allow us to execute queries recursively.

Note: keeping parameter in a string query statement:

**“select \* from where fisrt\_name = ?”**

? can be replaced with a parameter later.

Creating a statement with a query string :

Qs = **con.prepareStatement(query\_str)**

Setting the value to pre defined parameter

**Qs.setInt(1, 5)** : (1st ? will be placed with Integer value 5 )

Moving in lines (rows) of a resultset

Ex: go to next line

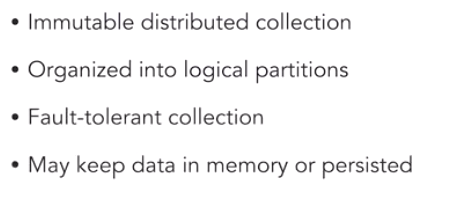
**resultSet.next**

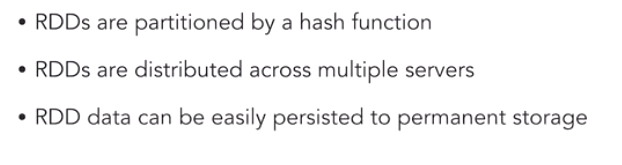
**resultSet.getString(“col\_name”);**

Note: Call getString(), getInt() depend on the sql data type of each column.

**Scala with Spark : Revision**

**RDDS:**





Creating a RDD

* Run Spark shell /usr/local/spark/bin
* Spark-shell
* Start spark program with scala

Ex: **Shuffle()** - Shuffle() given values

Then the created collection, can be converted to a RDD:

**Mapping Functions over RDD**

Creating RDD from shuffled collection:



**take()**

**rdd.take(25)** : get only 25 elements from RDD elements

**mean()**

* Get the mean of the RDD element distribution

**Statistics over RDD**

Package to import: o**rg.apache.spark.mllib.stat.Statistics**

**map()**  → rdd1.map( some function ) ex: .map( \_ \* 2 )

**rdd.takeSample( true, 1000 )**

[ true: withReplacement, 1000: no of elements to sample ]

Other statistic methods:

.mean

.min

.max

**.stats** (gives all common stats details)

**Array.fill(no of elements) (Random.nextDouble)**

Random → is from Random package. .nextDouble is to Double values

**Working with JSON Files**

Import sql session:

* org.apache.spark.sql.Sparksession
* Val Spark - sparkSession.builder().appName(’app\_name’).geOrCreate()
* spark.read.json(“path\_to\_json\_file”);