Program :-1

1. Introduction to Scala and Spark

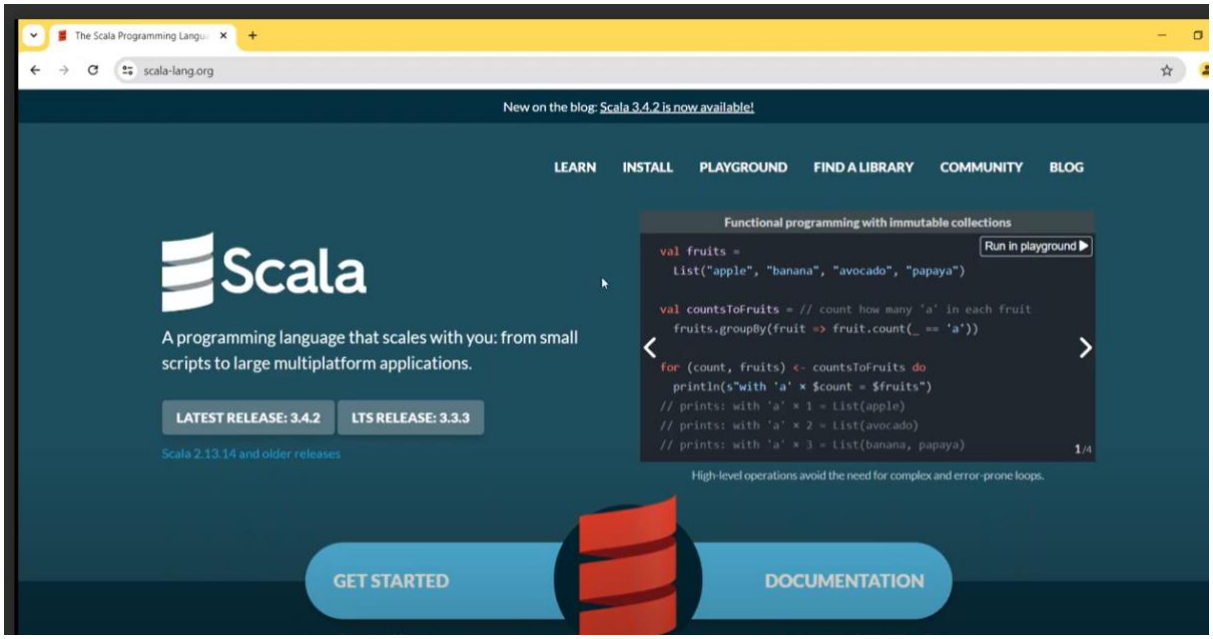
i. Install Scala, Spark, and set up the development environment

ii. Explore basic Scala syntax, variables, and data types

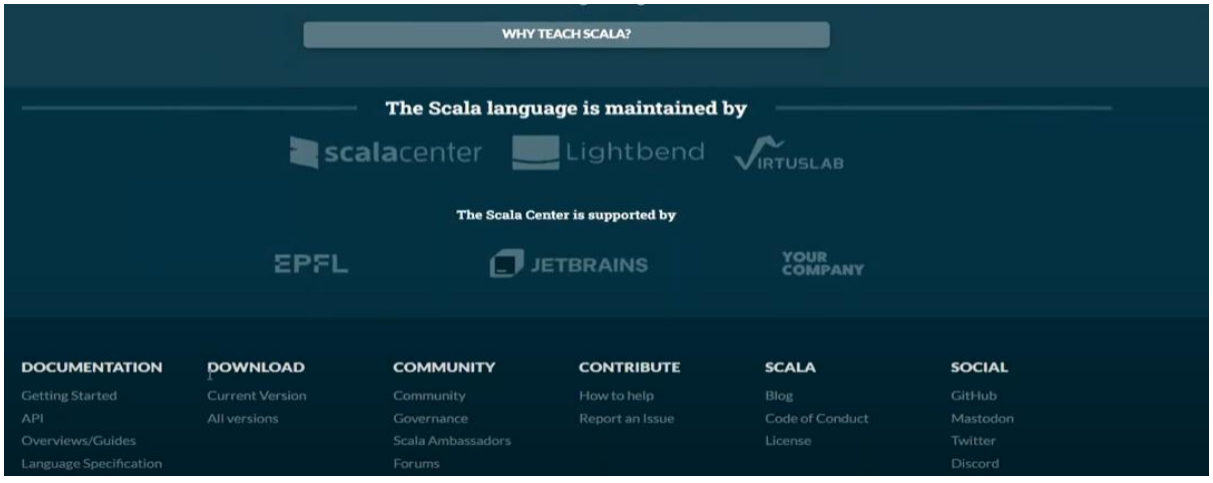
iii. Use Spark's interactive shell to execute basic Spark commands in Scala.

STEP 1: TO INSTALL ANY VERSION OF SCALA , WE MUST REQUIRES “JAVA 1.6” OR ANY HIGHER VERSIOS TO BE INSTALLED WELL IN HAND.BECAUSE ,BOTH SCALA & JAVA RUNS ON TOP OF JVM(JAVA VIRTUAL MACHINE)

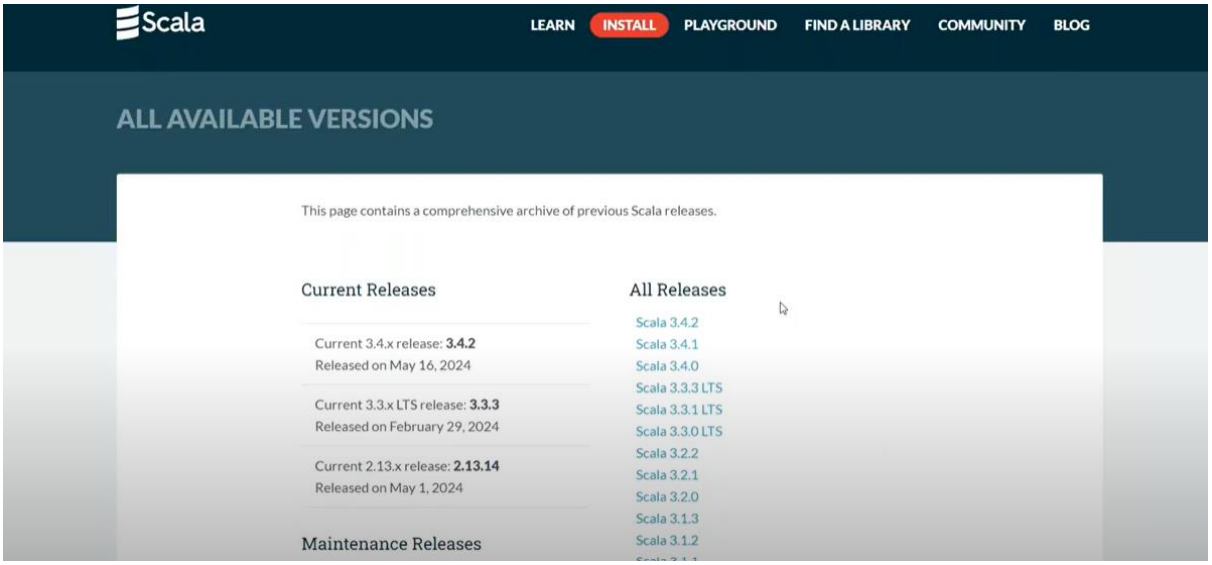
STEP 2: THE OFFICIAL WEBSITE OF SCALA: --------→//WWW.SCALA-LANG.ORG



STEP3:

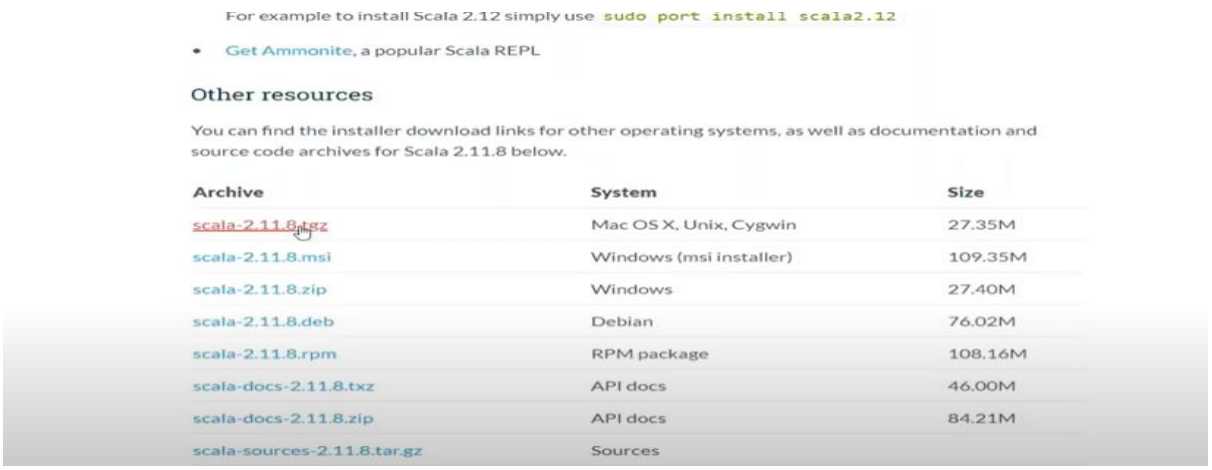


STEP4: CLICK ON All versions

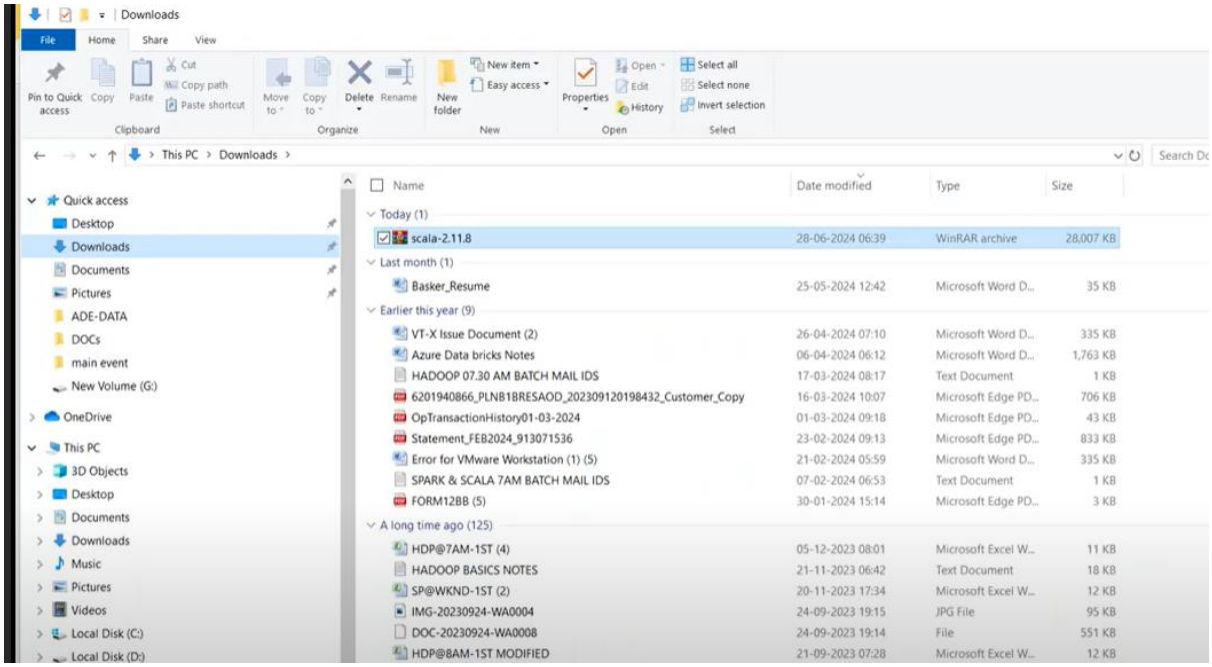


STEP 5: download scala 2.11.8 version

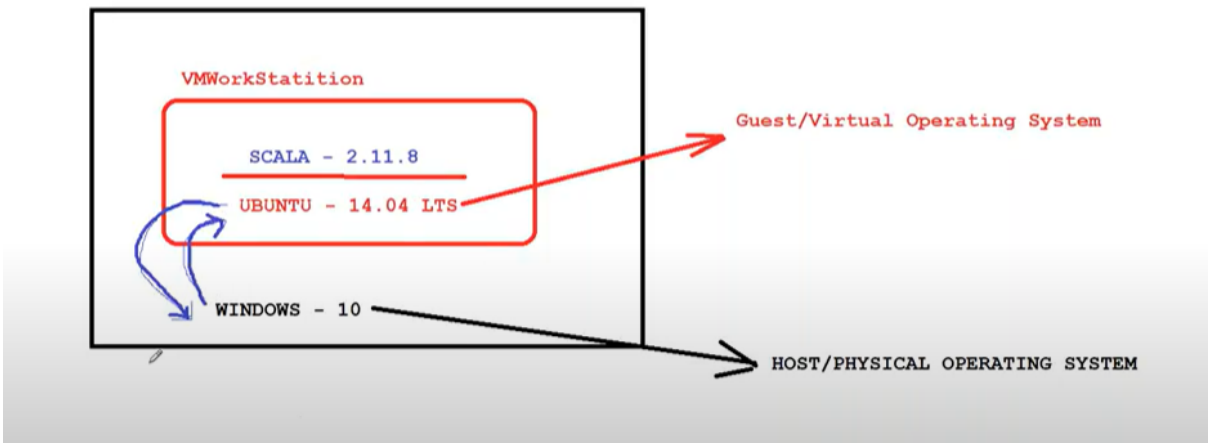
STEP 6:



STEP 7:



STEP 8:How exactly scala works in our system:



SCALA – 2.11.8 INSTALLATION GUIDE

STEP 1: Download scala-2.11.8.tgz from the below url.

http://scala-lang.org/download/2.11.8.html

```
gopalkrishna@ubuntu:~/INSTALL$ pwd
/home/gopalkrishna/INSTALL
gopalkrishna@ubuntu:~/INSTALL$ cp /home/gopalkrishna/Downloads/scala-2.11.8.tgz .
gopalkrishna@ubuntu:~/INSTALL$
gopalkrishna@ubuntu:~/INSTALL$
```

STEP 2: Copy the Downloaded TARBALL to 'INSTALL' directory [Path: /home/gopalkrishna/INSTALL]

STEP 3: Extract the copied tarball using below command:

tar -xzvf scala-2.11.8.tgz

```
gopalkrishna@ubuntu:~/INSTALL$
gopalkrishna@ubuntu:~/INSTALL$ ll scala-2.11.8.tgz
-rw-rw-r-- 1 gopalkrishna gopalkrishna 28678231 Mar 27 07:36 scala-2.11.8.tgz
gopalkrishna@ubuntu:~/INSTALL$
gopalkrishna@ubuntu:~/INSTALL$ tar -xzvf scala-2.11.8.tgz
```

STEP 4: After tarball extraction, we get below directory

scala-2.11.8/

```
gopalkrishna@ubuntu:~/INSTALL$ ll
total 725824
drwxrwxr-x 10 gopalkrishna gopalkrishna      4096 Mar 27 07:37 ./
drwxr-xr-x 28 gopalkrishna gopalkrishna      4096 Mar 27 02:52 ../
drwxrwxr-x  8 gopalkrishna gopalkrishna      4096 Mar 18 03:09 apache-hive-1.2.1-bin/
-rwxrw-rw-  1 gopalkrishna gopalkrishna  92834839 Feb 17 00:20 apache-hive-1.2.1-bin.tar.gz*
drwxr-xr-x 10 gopalkrishna gopalkrishna      4096 Mar 10 02:19 hadoop-2.6.0/
-rw-r--r--  1 gopalkrishna gopalkrishna 195257604 Mar 10 01:44 hadoop-2.6.0.tar.gz
drwxrwxr-x  8 gopalkrishna gopalkrishna      4096 Mar 22 03:16 hbase-1.1.2/
-rw-rw-r--  1 gopalkrishna gopalkrishna 102754526 Mar 22 03:11 hbase-1.1.2-bin.tar.gz
drwxrwxr-x  3 gopalkrishna gopalkrishna      4096 Mar 22 03:44 hbasedata/
-rw-------  1 gopalkrishna gopalkrishna    983911 Jan 28  2016 mysql-connector-java-5.1.38.jar
drwxr-xr-x 16 gopalkrishna gopalkrishna      4096 Jun  1  2015 pig-0.15.0/
-rwxrw-rw-  1 gopalkrishna gopalkrishna 120917625 Dec 12  2015 pig-0.15.0.tar.gz*
drwxrwxr-x  6 gopalkrishna gopalkrishna      4096 Mar  4  2016 scala-2.11.8/
-rw-rw-r--  1 gopalkrishna gopalkrishna  28678231 Mar 27 07:36 scala-2.11.8.tgz
drwxr-xr-x 12 gopalkrishna gopalkrishna      4096 Sep 28 17:04 spark-2.0.1-bin-hadoop2.6/
-rw-rw-r--  1 gopalkrishna gopalkrishna 184890838 Mar 27 02:31 spark-2.0.1-bin-hadoop2.6.tgz
drwxr-xr-x  9 gopalkrishna gopalkrishna      4096 Apr 26  2015 sqoop-1.4.6.bin__hadoop-2.0.4-alpha/
-rw-rw-r--  1 gopalkrishna gopalkrishna  16870735 Mar 22 01:50 sqoop-1.4.6.bin__hadoop-2.0.4-alpha.ta
```

STEP 5: Update the SCALA_HOME & PATH

Variables in bashrc file (command: nano ~/.bashrc)

```
# Below 2 lines we have to add for SCALA Installation
export SCALA_HOME=/home/gopalkrishna/INSTALL/scala-2.11.8
```

export PATH=$PATH:$SCALA_HOME/bin

```
  GNU nano 2.2.6          File: /home/gopalkrishna/.bashrc                Modified


# Below 2 lines we have to add for SPARK Installation

export SPARK_HOME=/home/gopalkrishna/INSTALL/spark-2.0.1-bin-hadoop2.6
export PATH=$PATH:$SPARK_HOME/bin

# Below 2 lines we have to add for SCALA Installation

export SCALA_HOME=/home/gopalkrishna/INSTALL/scala-2.11.8
export PATH=$PATH:$SCALA_HOME/bin
```

STEP 6: To check the bashrc changes, open a new terminal and type 'echo $SCALA_HOME' command

```
gopalkrishna@ubuntu:~$ echo $SCALA_HOME
/home/gopalkrishna/INSTALL/scala-2.11.8
gopalkrishna@ubuntu:~$
```

STEP 7: After Successful installation of SCALA, to get into SCALA prompt, use below command from any path

Scala

```
gopalkrishna@ubuntu:~$ scala
Welcome to Scala 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.7.0_121).
Type in expressions for evaluation. Or try :help.

scala>
```

STEP 8: To Know the Installed Version of SCALA

```
gopalkrishna@ubuntu:~$ scala -version
Scala code runner version 2.11.8 -- Copyright 2002-2016, LAMP/EPFL
gopalkrishna@ubuntu:~$
```

STEP 9: To come out of scala prompt

```
scala> :q
gopalkrishna@ubuntu:~$
```

To Print a message at SCALA Prompt

```
gopalkrishna@ubuntu:~$ scala
Welcome to Scala 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.7.0_121).
Type in expressions for evaluation. Or try :help.

scala> println("Hello Scala")
Hello Scala

scala> print("Hello Scala")
Hello Scala
scala>
```

HOW TO DECLARE VARIABLES IN SCALA:

var(variable) OR val(value):

1.var(variable) ---> is a mututable entity (at any time , we can change the value of a variable which is declared with VAR)

Syntax: var <<variableName>>:<<datatype>> = <<valueofvariable>>

Ex: 1. var x:Int = 10

x:Int = 10

print(x) ---> 10

2. x = 20

x:Int = 20

print(x) ---> 20

2. val(value) ---> Immutable Entity(Non-Changeble Entity)

1. **val y:Int = 50**

   y:Int = 50

   print(y) ---> 50

2. **y = 60**

   ERROR: reassignment to val

SPARK INSTALLATION GUIDE

Pre-Requisites for Spark Installation:

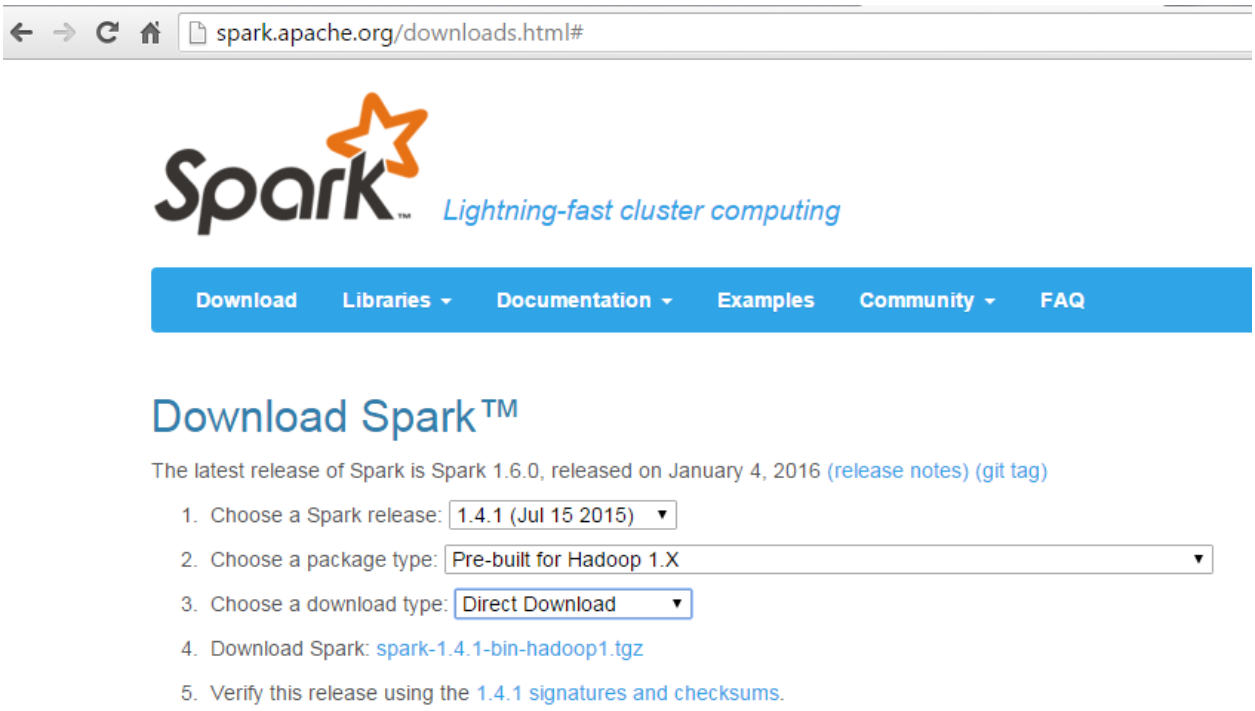
As Spark itself written in a language called "Scala" and

Scala language can only runs on JVM(Java Virtual Machine) which requires Java 6 or the above version,

For Spark Installation also → Java 6 or above version must be installed on the machine.

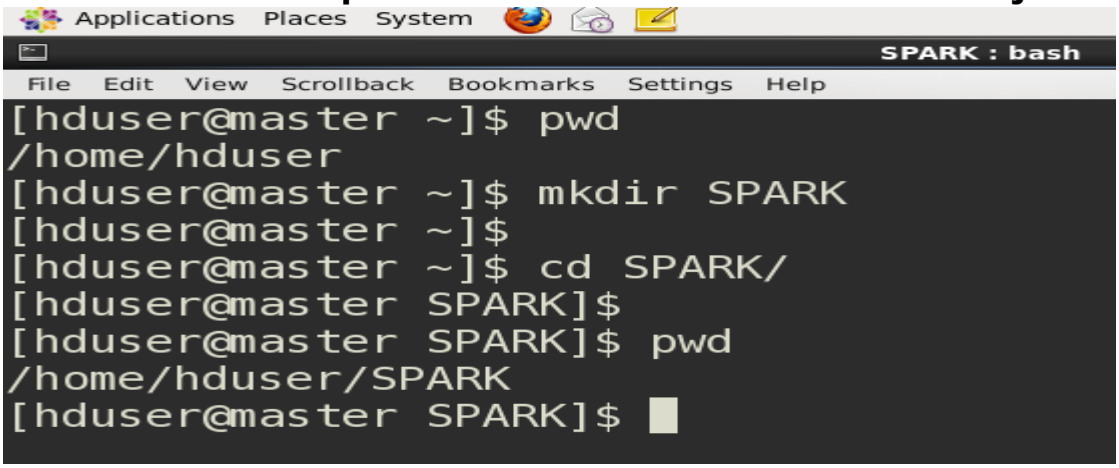
STEP 1: Go to website: http://spark.apache.org/downloads.html

STEP 2: Choose a Spark release: 1.4.1 & Choose a package type as: Pre-built for Hadoop 1.x

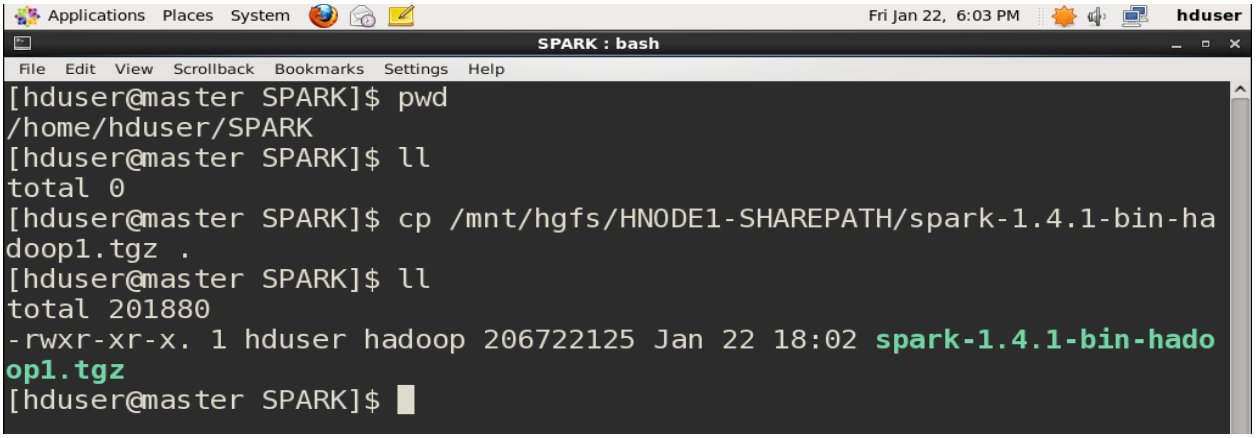


STEP 3: Click on Download Spark Link (.tgz)

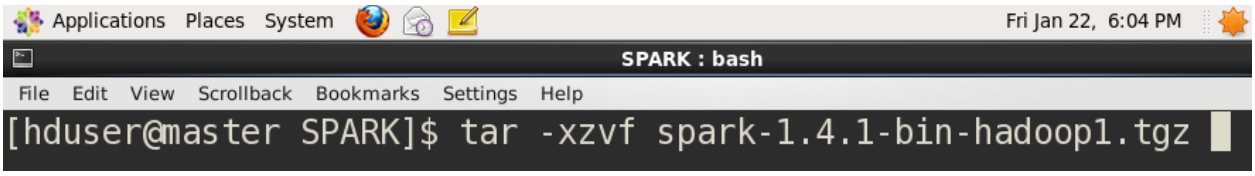
STEP 4: Go to any of the Linux terminal (Ubuntu OR Cent OS) and Create Folder for Spark Installation and Check installed java version

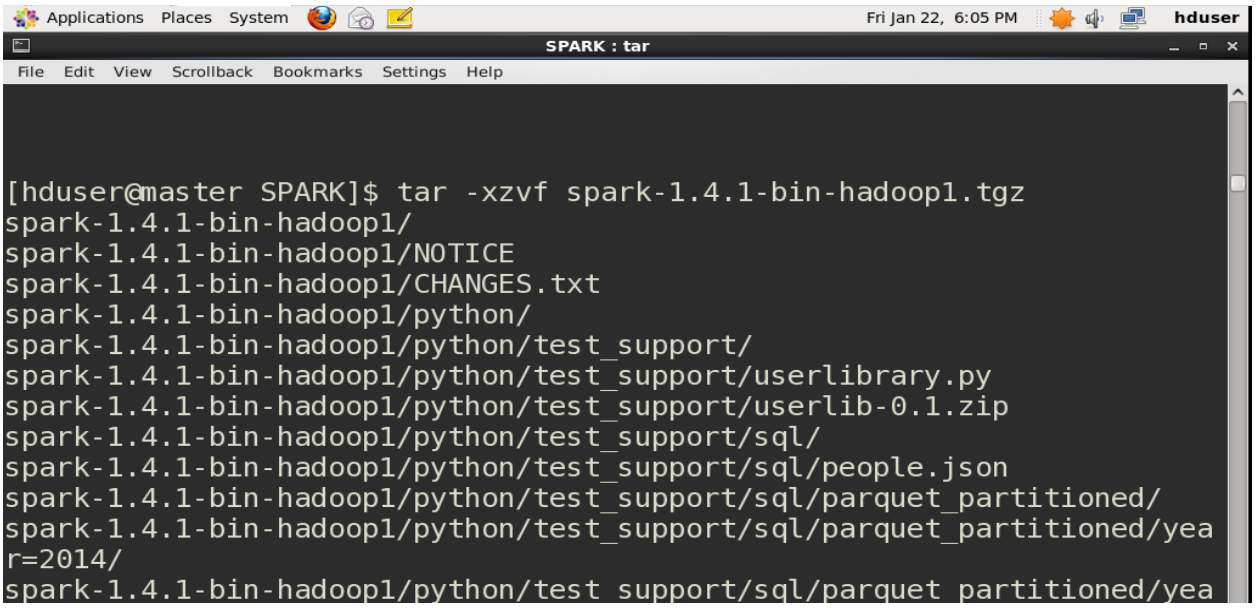


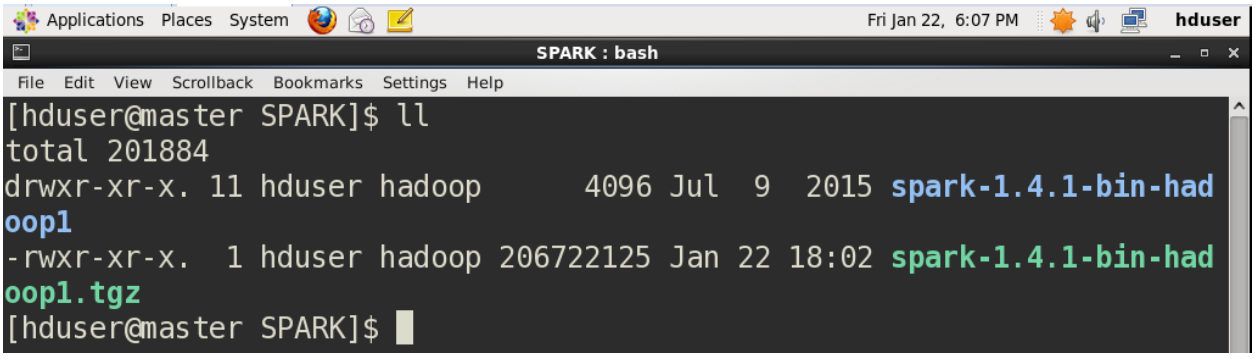
STEP 4: Place the Downloaded Tarball in Share Path Of the VMWorkStation and Copy the same Tarball to the Spark Directory (Which is Created in the STEP 3)



STEP 5: Extract the Tarball



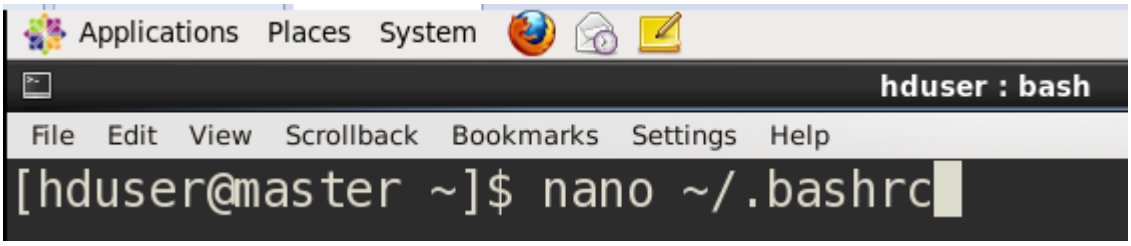


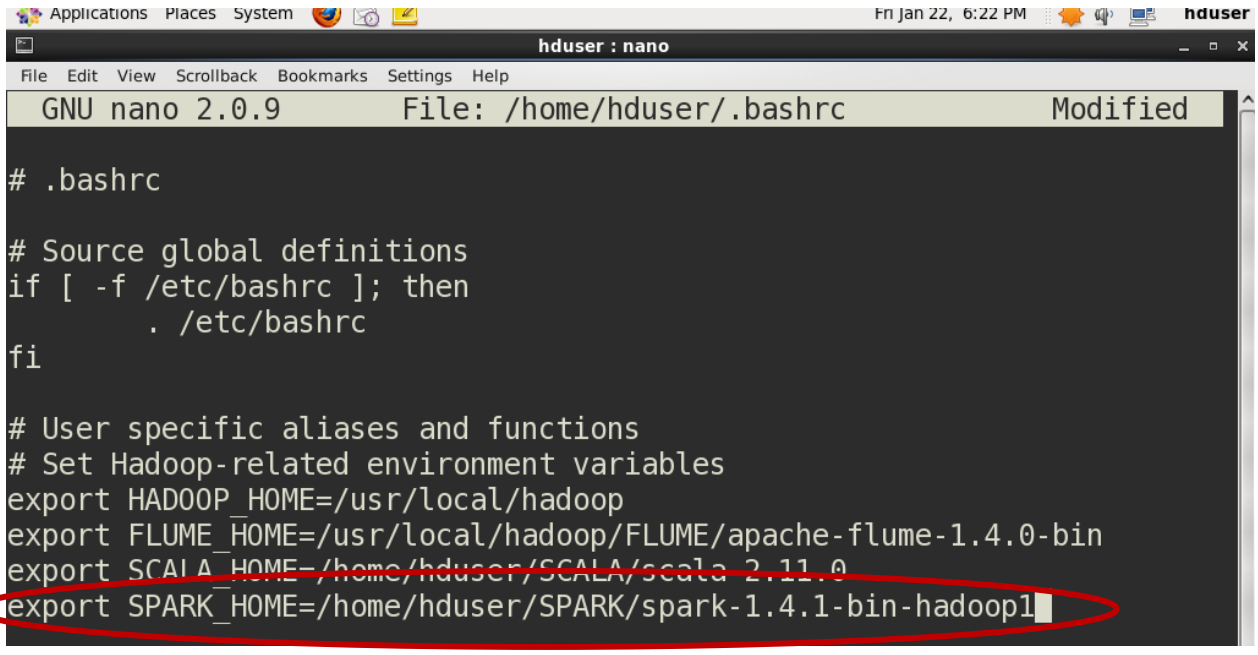


STEP 6: Navigate to the spark-1.4.1-bin-hadoop1 directory

```
[hduser@master SPARK]$ cd spark-1.4.1-bin-hadoop1
[hduser@master spark-1.4.1-bin-hadoop1]$
[hduser@master spark-1.4.1-bin-hadoop1]$ ll
total 692
drwxr-xr-x. 2 hduser hadoop   4096 Jul  9  2015 bin
-rw-r--r--. 1 hduser hadoop 584344 Jul  9  2015 CHANGES.txt
drwxr-xr-x. 2 hduser hadoop   4096 Jul  9  2015 conf
drwxr-xr-x. 3 hduser hadoop   4096 Jul  9  2015 data
drwxr-xr-x. 3 hduser hadoop   4096 Jul  9  2015 ec2
drwxr-xr-x. 3 hduser hadoop   4096 Jul  9  2015 examples
drwxr-xr-x. 2 hduser hadoop   4096 Jul  9  2015 lib
-rw-r--r--. 1 hduser hadoop  50971 Jul  9  2015 LICENSE
-rw-r--r--. 1 hduser hadoop  22559 Jul  9  2015 NOTICE
drwxr-xr-x. 6 hduser hadoop   4096 Jul  9  2015 python
drwxr-xr-x. 3 hduser hadoop   4096 Jul  9  2015 R
-rw-r--r--. 1 hduser hadoop   3624 Jul  9  2015 README.md
```

STEP 7: Add the SPARK_HOME in bashrc file

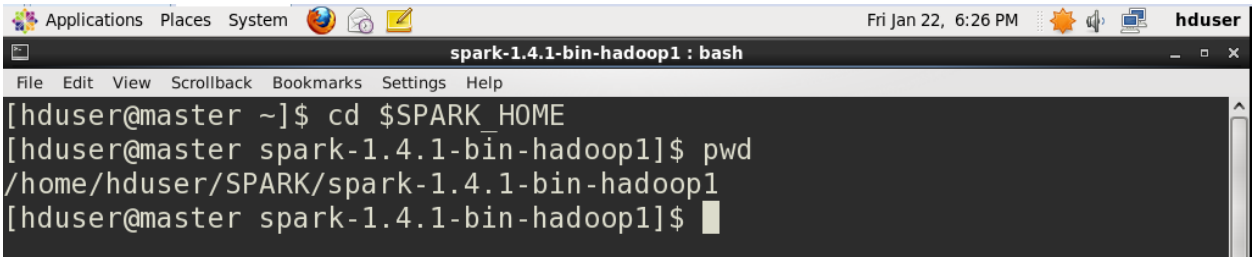




Also Add the SPARK_HOME/bin → to PATH variable

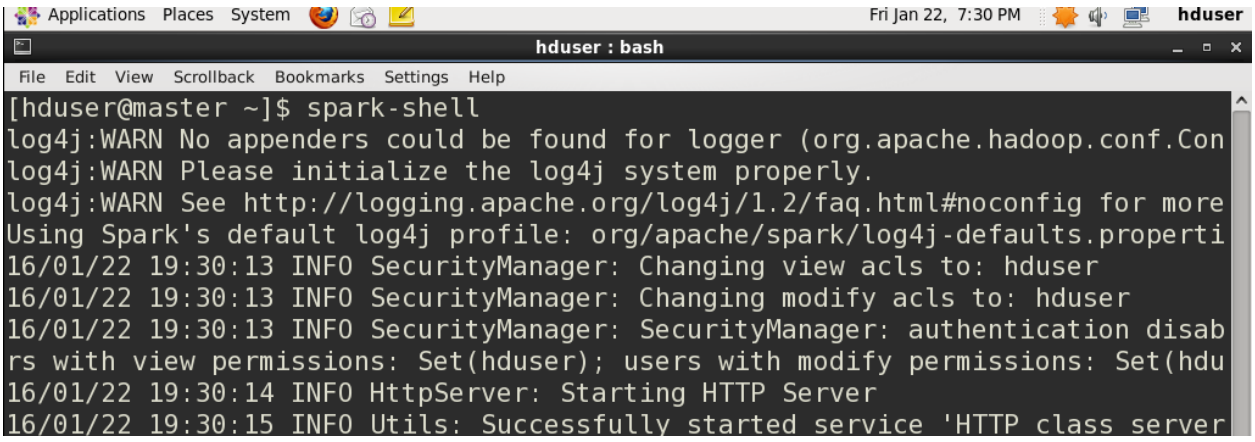
```
# Add Hadoop bin/ directory to PATH
export PATH=$PATH:$HADOOP_HOME/bin:$PATH:$JAVA_HOME/bin:$PATH:$FLUME_H$
export PATH=$PATH:$SCALA_HOME/bin
export PATH=$PATH:$SPARK_HOME/bin
```

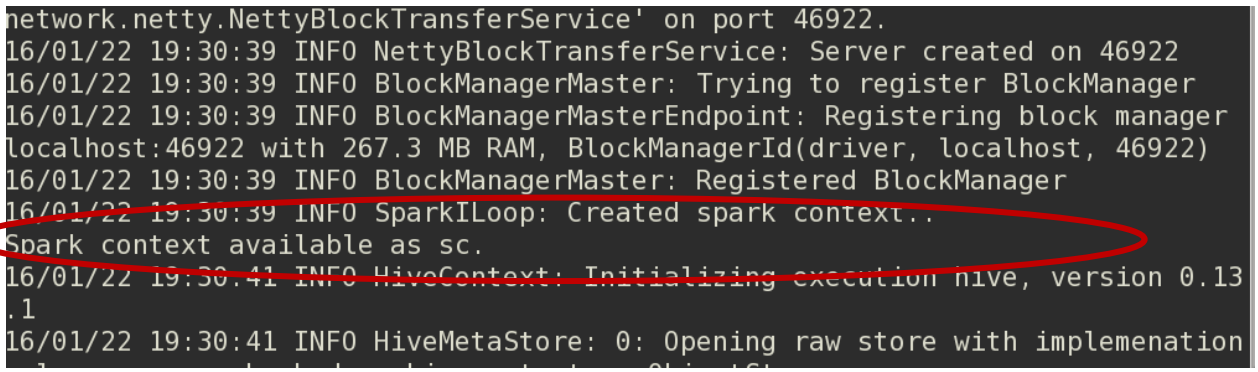
STEP 8: Open A New Terminal and Check whether bashrc Changes are reflecting or not



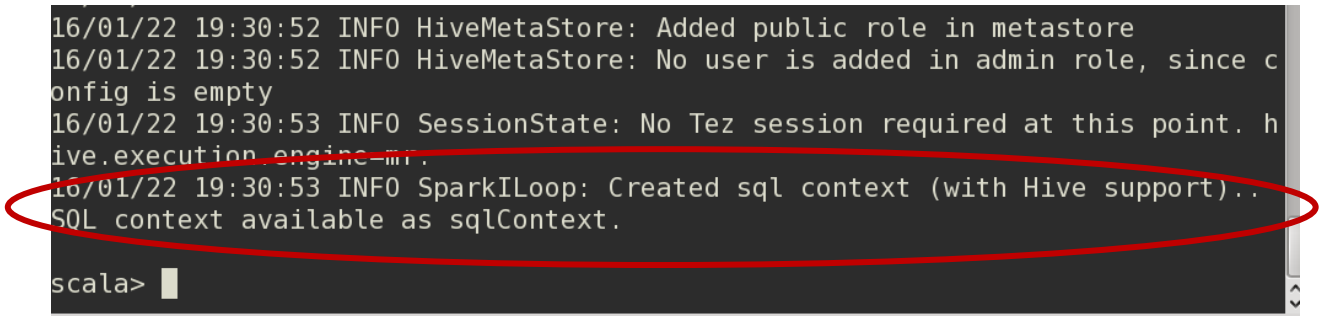
STEP 9: From any path, type the below command to enter Into the Spark Shell (through Scala)

Spark-shell





NOTE: Default Spark Context is "sc"



NOTE: Default SQL Context is "sqlContext"

==========

DATA TYPES IN SCALA :

Scala is a statically typed language, meaning that the type of a variable is known at compile time. Scala supports a wide range of data types, divided into two main categories:

1. **Value Types**: These are the basic types of data like numbers, characters, and Booleans.
2. **Reference Types**: These include classes, arrays, and other complex structures.

Basic Data Types:

1. Byte

- **Description**: 8-bit signed integer.
- **Range**: -128 to 127.
- **Example**: val byteVal: Byte = 100

2. Short

- **Description**: 16-bit signed integer.

- **Range**: -32,768 to 32,767.
- **Example**: val shortVal: Short = 20000

3. Int

- **Description**: 32-bit signed integer.
- **Range**: -2^31 to 2^31-1.
- **Example**: val intVal: Int = 100000

4. Long

- **Description**: 64-bit signed integer.
- **Range**: -2^63 to 2^63-1.
- **Example**: val longVal: Long = 10000000000L

5. Float

- **Description**: 32-bit IEEE 754 floating-point number.
- **Range**: Approximately ±1.4E-45 to ±3.4E38.
- **Example**: val floatVal: Float = 10.5F

6. Double

- **Description**: 64-bit IEEE 754 floating-point number (default for floating-point numbers).
- **Range**: Approximately ±4.9E-324 to ±1.7E308.
- **Example**: val doubleVal: Double = 20.99

7. Char

- **Description**: 16-bit unsigned Unicode character.
- **Range**: 0 to 65,535 (can represent any character in the Unicode standard).
- **Example**: val charVal: Char = 'A'

8. String

- **Description**: A sequence of characters (not a primitive type but commonly used).
- **Example**:val stringVal: String = "Hello, Scala!"

9. Boolean

- **Description**: Represents a value of either true or false.
- **Example**:val booleanVal: Boolean = true

10. Unit

- **Description**: Represents no value, similar to void in Java. The only value of this type is ().
- **Example**:

```
def printMessage() {
println("This returns Unit")}
```

11. Null

- **Description**: A reference type, representing a null or non-existent value. Can be assigned to any reference type.
- **Example**:

```
val nullVal: String = null
```

12. Nothing

- **Description**: The bottom type; it is a subtype of every other type. It's used to indicate abnormal termination, such as an exception.
- **Example**:

```
def throwError(): Nothing = throw new RuntimeException("Error!")
```

13. Any

- **Description**: The root of Scala's type hierarchy. Every type in Scala is a subtype of Any.
- **Example**:val anyVal: Any = 123 // Can hold any type of value

14. AnyVal

- **Description**: The root of all value types (Byte, Short, Int, Long, Float, Double, Char, Boolean, Unit).
- **Example**:val anyVal: AnyVal = 10

15. AnyRef

- **Description**: The root of all reference types (all classes, objects, arrays, etc.). Equivalent to Object in Java.

- **Example**:

  val anyRefVal: AnyRef = "Scala String"

What is "Type Inference in Scala" ?

In variable declaration of Scala , though we are not providing the DATATYPE information , based on right hand value of the variable Scala will automatically detects the corresponding datatype info...this feature is known as "Type Inference".

Ex : x:Int = 10

x:Int = 10

Type Inference Sign

x = 10

x:Int = 10

Write and execute a Scala Program:

1. **Create a Scala File**:
   - Open your preferred text editor or IDE.(nano HelloWorld.scala)
   - Create a new Scala file, e.g., HelloWorld.scala:

```
object HelloWorld
{
  def main(args: Array[String])
 {
   println("Hello, World!")
  }
}
```

Explanation:

1. **object HelloWorld**:
   - This line declares a **singleton object** named HelloWorld.
   - In Scala, object is used to define a singleton, meaning there is only one instance of HelloWorld in the program. It's similar to a class in other languages but without the need to instantiate it.
2. **def main(args: Array[String]): Unit**:
   - This line defines the main method, which is the entry point of the Scala application.

- **def** is used to define a method. The method is named main, and it takes an argument args which is an array of strings (Array[String]).
- The method returns Unit, which is similar to void in Java. It means that the method does not return any value.

3. **println("Hello, World!")**:
   - This line prints the string "Hello, World!" to the console.
   - **println** is a standard output function in Scala used to print text followed by a newline.

How It Works:

- When you run this Scala program, the Scala compiler looks for an object with a main method that takes an array of strings as an argument.
- The main method is executed, which contains the println("Hello, World!") statement.
- As a result, the text "Hello, World!" is printed to the console.

Save the file.

2. **Compile the Scala Program**:
   - Open the terminal and navigate to the directory containing your Scala file.
   - Compile the program using the scalac command:
     scalac HelloWorld.scala
   - This command generates a HelloWorld.class file.

3. **Run the Scala Program**:
   - Execute the compiled Scala program using the scala command:
     scala HelloWorld
   - You should see the output:
     Hello, World!

PROGRAM 2:

Scala Functional Programming

i. Define and use functions in Scala

ii. Explore higher-order functions and their applications

iii. Work with immutable collections (List, Set, Map) in Scala

iv. Perform common operations using Scala's functional programming style

SCALA LOOPS

(while loop , do..while loop, for loop)

While loop

A while loop statement repeatedly executes a target statement as long as a given condition is true.

Syntax:

```
while(condition)
{
  statement(s);
}
```

Example:

```
object whileLoopPrac
 {

   def main(args: Array[String])
    {


     var a = 10;// local variable
      // while loop execution
      while( a < 20 )
      {

          println( "Value of a: " + a );
          a = a + 1;
      }

    }

}
```

Do…While loop

Unlike **while loop**, which **tests the loop condition at the top of the loop**, the **do...while loop** checks its condition **at the bottom of the loop**.

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute **at least one time**.

Syntax:

```
Do
{
   statement(s);
}while( condition );
```

Example:

```
object doWhileLoopPrac
{
    def main(args: Array[String])
  {
        var a = 10;
    // Do while loop execution
      do
      {
         println( "Value of a: " + a );
         a = a + 1;
      } while (a > 20)
  }
}
```

for loop

A **for loop** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Various forms of for loop in Scala:

- ➔ **For Loop with Range**
- ➔ **For Loop with Collections**
- ➔ **For Loop with Filters**
- ➔ **For Loop with Yield**

1. **For Loop with Range:**

```
for( var x <- Range )
{
```

```
    statement(s);
}
```

NOTE: Here Range is the number of numbers that reprasendted as I to J OR I unti J

NOTE: Here "<- " Is known as "Generator" command as this operator only generating individual values from range of numbers..

Example: using "I to J"

```
object forLoopPrac
{

  def main(args: Array[String])
  {

    var a = 0; //
    //for loop execution with a range ,    "<-" is a generator operator
   for( a <- 1 to 10)
   {

     println( "Value of a: " + a );
   }

  }

}
```

Example 2: using "I until J"

```
object forLoopPrac_Until
{

  def main(args: Array[String])
  {

    var a = 0;
    // until 10 means..10 is NOT included up to 9 only
    for( a <- 1 until 10)
    {

       println( "Value of a: " + a );
    }

  }

}
```

2. For Loop with Collection

```
for( var x <- List )
{

    statement(s);
}
```

NOTE: Here, the List variable is a collection type having a list of elements and for loop iterate through all the elements returning one element in x variable at a time

Example:

```
object forLoopPrac_Collction
{
```

```
  def main(args: Array[String])
  {

     var a = 0;
     val numList = List(1,2,3,4,5,6);

     for( a <- numList)
     {

        println( "Value of a: " + a );
     }

  }

}
```

3. For Loop with Filter

```
for( var x <- List
        if condition1; if condition2...
  )

{

     statement(s);
}
```

Example:

```
object forLoopPrac_Filter
{

  def main(args: Array[String])
  {

     var a = 0;
     val numList = List(1,2,3,4,5,6,7,8,9,10);

     for( a <- numList if a != 3; if a < 8 )
     {

        println( "Value of a: " + a );
     }

  }

}
```

```
Output:

Value of a: 1
Value of a: 2
Value of a: 4
Value of a: 5
Value of a: 6
Value of a: 7
```

4. For Loop with Filter:

To store return values from a for loop in a variable or can return through a function. To do so, you prefix the body of the for expression by the keyword yield as follows

```
var retVal = for{ var x <- List
if condition1; if condition2...
} yield x

object forLoopPrac_Yield
```

```
{

  def main(args: Array[String])
    {

      var a = 0;
      val numList = List(1,2,3,4,5,6,7,8,9,10);
     // for loop execution with a yield

     var retVal = for{ a <- numList if a != 3; if a < 8 }yield a

    // Now print returned values using another loop.

    for( a <- retVal)
    {

println( "Value of a: " + a );
    }

}
}
```

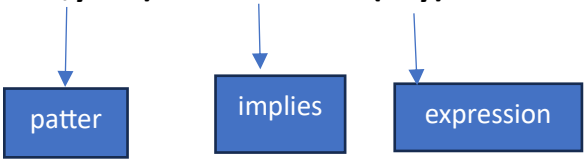
SCALA FUNCTIONS

1.Scala closure (or)scala named functions

2.Higher order Functions[foreach(),map(),reduce()]

3.Currying Functions.

Scala>Var fun1= (x:Int,y:Int) => (x+y)



Fun1: (Int,Int)=> Int=<function>

Return type is integer

Return value is sum of x+y

1.Scala Closure is a function, whose **return value depends** on the value of one or more variables **declared outside this function**.

```
scala> var fun1=(x:Int,y:Int) => (x+y);
fun1: (Int, Int) => Int = <function2>

scala> println("output is :" +fun1(4,6));
output is :10

scala>
```

- **For example if I want the value in double**

```
scala> var fun1=(x:Int,y:Int) => (x+y).toDouble;
fun1: (Int, Int) => Double = <function2>

scala> println("output is :" +fun1(4,6));
output is :10.0

scala>
```

```
scala> var fun1 = (i:Int) => i*10;
fun1: Int => Int = <function1>

scala> println(" Result is: " + fun1(2));
 Result is: 20

scala> println(" Result is: " + fun1(1));
 Result is: 10

scala> println(" Result is: " + fun1(0));
 Result is: 0

scala>
```

NOTE: Here "i" is defined as parameter to the function (**fun1()**)..depending on " i" value which we are passing→ result is varying ..

EXAMPLE 2:

```
scala> var gkfun=(x:String,y:String,z:String) => x + "\t" + y.reverse + "|" + z.length
gkfun: (String, String, String) => String = <function3>

scala> println("Result is:" +gkfun("spark","scala","Ruby");
<console>:1: error: ')' expected but ';' found.
println("Result is:" +gkfun("spark","scala","Ruby");
                                                  ^

scala> println("Result is:" +gkfun("spark","scala","Ruby"));
Result is:spark alacs|4
```

```
scala> var fun2 = (x:String) => x + "Scala";
fun2: String => String = <function1>

scala> println("Result String Is: " + fun2(" Spark "));
Result String Is:  Spark Scala

scala> println("Result String Is: " + fun2(" Java,Phython and  "));
Result String Is:  Java,Phython and  Scala

scala> println("Result String Is: " + fun2(null));
Result String Is: nullScala

scala> println("Result String Is: " + fun2(" "));
Result String Is:  Scala

scala>
```

```
scala> println("Result String Is: " + fun2());
<console>:9: error: not enough arguments for method apply: (v1: String)String in trai
 Function1.
Unspecified value parameter v1.
              println("Result String Is: " + fun2());
                                                 ^

scala>
```

HIGHER ORDER FUNCTIONS IN SCALA

[foreach() , map() , reduce()]

1.Foreach() - > example

```
scala> val courses = List("Spark","Scala","Hadoop","Kafka");
courses: List[String] = List(Spark, Scala, Hadoop, Kafka)

scala> courses.foreach( (c:String) => (println(c)) );
Spark
Scala
Hadoop
Kafka

scala>
```

NOTE: foreach() takes a function (a procedure, to be accurate) and invokes it with every item in the list

Alternative way of invoking every item in the list

```
scala> val cSize = courses.map( (c) => (c.size) );
cSize: List[Int] = List(5, 5, 6, 5)

scala> println( courses(0)  + "\t" + cSize(0));
Spark   5

scala> println( courses  + "\t" + cSize);
List(Spark, Scala, Hadoop, Kafka)       List(5, 5, 6, 5)

scala>
```

```
scala> val courses = List("Spark","Scala","Hadoop","Kafka");
courses: List[String] = List(Spark, Scala, Hadoop, Kafka)

scala> courses.foreach( (c:String) => (println(c)) );
Spark
Scala
Hadoop
Kafka

scala> courses.foreach( (c) => (println(c)) );
Spark
Scala
Hadoop
Kafka

scala>
```

2.Map() -> Example

```
scala> val courses = List("Spark","Scala","Hadoop","Kafka");
courses: List[String] = List(Spark, Scala, Hadoop, Kafka)

scala> val cSize = courses.map( (c) => (c.size) );
cSize: List[Int] = List(5, 5, 6, 5)

scala> println(cSize);
List(5, 5, 6, 5)
```

NOTE: map() takes a function that converts a single list element to another value and/or type.

3.Reduce() -> Example

```
scala> val marks = List(20,30,40,50,60);
marks: List[Int] = List(20, 30, 40, 50, 60)

scala> val totalMarks = marks.reduce( (a:Int,b:Int) => (a+b) );
totalMarks: Int = 200

scala> val totalMarks = marks.reduce( (a,b) => (a+b) );
totalMarks: Int = 200

scala>
```

NOTE: reduce () takes a function that combines two list elements into a

single element

SCALA ARRAYS

To create an array of integers

scala> var arrayElements = Array(1,2,3,4,5,6,7,8,9,10)

arrayElements: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

To Print the Elements of an array

scala> for(x <- arrayElements) { println(" Array Elements: " + x) } scala>

println(arrayElements.length)
10

scala> println(arrayElements(7)) 8

To make the SUM of Array Elements

scala> var total=0;for (i <- 0 to (arrayElements.length - 1)) { total += arrayElements(i); }
total: Int = 55 scala>

To create an array of strings

scala> var arrayEle = Array("AAA","BBB","CCC","DDD","EEE")
arrayEle: Array[String] = Array(AAA, BBB, CCC, DDD, EEE)

scala> println(arrayEle(2))
CCC

scala> println(arrayEle.length) 5

SCALA COLLECTIONS

List: Scala Lists are quite similar to arrays which means, all the elements of a list have the same type, but there are two important differences. First, **lists are immutable**, which means elements of a list cannot be changed by assignment.

Second, lists **represent a linked list whereas arrays are flat**.

```
scala> val names: List[String] = List("Ram", "Lakshman", "Seetha");
names: List[String] = List(Ram, Lakshman, Seetha)

scala> val names: List[String] = List("RamChandra", "Lakshman", "Seetha");
names: List[String] = List(RamChandra, Lakshman, Seetha)

scala>  names = List("SriRamChandra", "Lakshman", "Seetha");
<console>:8: error: reassignment to val
        names = List("SriRamChandra", "Lakshman", "Seetha");
              ^
```

To Access the List Elements

```
scala> println(names(0));
RamChandra

scala> println(names(1));
Lakshman

scala> println(names(2));
Seetha

scala>
```

```
scala> val ranks: List[Int] = List(1, 2, 3, 4)
ranks: List[Int] = List(1, 2, 3, 4)

scala> val ranks = List(1,2,3,4)
ranks: List[Int] = List(1, 2, 3, 4)

scala>
```

Operations on Lists: head, tail & isEmpty

```
scala> val ranks = List(1,2,3,4)
ranks: List[Int] = List(1, 2, 3, 4)

scala> println(ranks.head);
1

scala> println(ranks.tail);
List(2, 3, 4)

scala> println(ranks.isEmpty);
false

scala>
```

All lists can be defined using two fundamental building blocks

1. a tail **Nil**
2. **::**

This is pronounced cons

Example:

val names = "Rama" :: ("Lakshman" :: ("Seetha" :: Nil))

```
scala> val names = "Rama" :: ("Lakshman" :: ("Seetha" :: Nil))
names: List[String] = List(Rama, Lakshman, Seetha)

scala>
```

To Concatenate the Two Lists (:::)

```
scala> val names = "Rama" :: ("Lakshman" :: ("Seetha" :: Nil))
names: List[String] = List(Rama, Lakshman, Seetha)

scala> val addresses = "Hyderabad" :: ("Bangalore" :: ("Chennai" :: Nil))
addresses: List[String] = List(Hyderabad, Bangalore, Chennai)
```

```
scala> var name_address = names ::: addresses
name_address: List[String] = List(Rama, Lakshman, Seetha, Hyderabad, Bangalore, Chenn
i)

scala>
```

Alternative: (List.concat (list1, list2))

```
scala> var name_address = List.concat(names,addresses)
name_address: List[String] = List(Rama, Lakshman, Seetha, Hyderabad, Bangalore, Chenn
i)

scala>
```

List.fill() method creates a **list consisting of zero or more copies** of the same element as follows:

```
gopal@ubuntu:~/SCALA$ cat listPrg.scala
object listPrg
{
  def main(args: Array[String])
   {
     val tech = List.fill(5)("Spark") // Repeats Spark for 5 times.
     println( "Technology : " + tech )
     val num = List.fill(6)(2)  // Repeats num 2 for 6 times.
     println( "num : " + num )
   }
}
gopal@ubuntu:~/SCALA$
```

```
gopal@ubuntu:~/SCALA$ nano listPrg.scala
gopal@ubuntu:~/SCALA$ scalac listPrg.scala
gopal@ubuntu:~/SCALA$
gopal@ubuntu:~/SCALA$ scala listPrg
Technology : List(Spark, Spark, Spark, Spark, Spark)
num : List(2, 2, 2, 2, 2, 2)
gopal@ubuntu:~/SCALA$
```

Partition: splits a list based on where it falls with respect to a predicate function.

```
scala> val ranks = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
scala> val data = List("Raj","Boby","anandbabu","Danya","rajamouli.s.s");
data: List[String] = List(Raj, Boby, anandbabu, Danya, rajamouli.s.s)

scala> println(data.sorted)
List(Boby, Danya, Raj, anandbabu, rajamouli.s.s)

scala> println(data.sortBy( a => a.length) )
List(Raj, Boby, Danya, anandbabu, rajamouli.s.s)

scala> println(data.sortWith( (a,b)  => a.length > b.length) )
List(rajamouli.s.s, anandbabu, Danya, Boby, Raj)

scala> println(data.sortWith( (a,b)  => a(0) > b(0)) )
List(rajamouli.s.s, anandbabu, Raj, Danya, Boby)

scala> val data = List("Raj","Boby","anandbabu","Danya","rajamouli.s.s");
data: List[String] = List(Raj, Boby, anandbabu, Danya, rajamouli.s.s)

scala> data.last
res8: String = rajamouli.s.s

scala> data.updated(0,"RajMohan")
res9: List[String] = List(RajMohan, Boby, anandbabu, Danya, rajamouli.s.s)

scala> data.updated(2,"Anand")
res10: List[String] = List(Raj, Boby, Anand, Danya, rajamouli.s.s)
```

Other Operations on Lists:

ZIP Option:

```
scala> val list1 = List(10,20,30,40)
list1: List[Int] = List(10, 20, 30, 40)

scala> val list2 = List("AAA","BBB","CCC","DDD")
list2: List[String] = List(AAA, BBB, CCC, DDD)

scala> val newList = list1.zip(list2)
newList: List[(Int, String)] = List((10,AAA), (20,BBB), (30,CCC), (40,DDD))
```

```
scala> val newList = list2.zip(list1)
newList: List[(String, Int)] = List((AAA,10), (BBB,20), (CCC,30), (DDD,40))

scala> val newList = list1.zip(list1)
newList: List[(Int, Int)] = List((10,10), (20,20), (30,30), (40,40))

scala>
```

```
scala> newList(0)
res15: (Int, Int) = (10,10)

scala> newList(0)._1
res16: Int = 10

scala> newList(0)._2
res17: Int = 10

scala>
```

```
scala> val list1 = List(10,20,30,40)
list1: List[Int] = List(10, 20, 30, 40)

scala> val list2 = List("AAA","BBB")
list2: List[String] = List(AAA, BBB)

scala> val newList = list1.zip(list2)
newList: List[(Int, String)] = List((10,AAA), (20,BBB))

scala> newList.toMap
res18: scala.collection.immutable.Map[Int,String] = Map(10 -> AAA, 20 -> BBB)

scala>
```

```
scala> newList.foreach(println)
(10,AAA)
(20,BBB)

scala> newList.reverse.foreach(println)
(20,BBB)
(10,AAA)

scala> newList.map(a => a)
res21: List[(Int, String)] = List((10,AAA), (20,BBB))

scala>
```

Range

```
scala> 1 to 4
res22: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4)

scala> 1 until 4
res23: scala.collection.immutable.Range = Range(1, 2, 3)

scala> 1 until 4 by 3
res24: scala.collection.immutable.Range = Range(1)

scala>
```

Set: Set is a collection that contains no duplicate elements. There are two kinds of Sets, the **immutable** and the **mutable**. The difference between mutable and immutable objects is that when an object is immutable, the object itself can't be changed.

By default, Scala uses the immutable set

```
scala> val ranks = List(1,2,3,4,2,2,3,2)
ranks: List[Int] = List(1, 2, 3, 4, 2, 2, 3, 2)

scala> val ranks = Set(1,2,3,4,2,2,3,2)
ranks: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)

scala>
```

```
scala> val ranks = Set("Gopal","Ravi","Gopal","Krishna")
ranks: scala.collection.immutable.Set[String] = Set(Gopal, Ravi, Krishna)
```

```
gopal@ubuntu:~/SCALA$ cat setPrg.scala
object setPrg
gopal@ubuntu:~/SCALA$ scalac setPrg.scala
gopal@ubuntu:~/SCALA$
gopal@ubuntu:~/SCALA$ scala setPrg
Head of Technologies : Spark
Tail of Technologies : Set(Scala, Phython)
 Is Technology empty? : false
 Is Ranks empty? : true
gopal@ubuntu:~/SCALA$
```

Concatenating Sets: can use either **++** operator or **Set.++()** method to concatenate two or more sets.

```
gopal@ubuntu:~/SCALA$ cat setConcatePrg.scala
object setConcatePrg
 {
   def main(args: Array[String])
   {
     val car1 = Set("Benz","Audi","BMW");
     val car2 = Set("i10","i20");
     val car3 = Set("Swift","Santro","Creta");

    // var carset = car1 ++ car2 ++ car3;
     var carset = car1.++(car2).++(car3)
     println("Selected Cars : " + carset);
   }
}
gopal@ubuntu:~/SCALA$
```

```
gopal@ubuntu:~/SCALA$ scalac setConcatePrg.scala
gopal@ubuntu:~/SCALA$
gopal@ubuntu:~/SCALA$ scala setConcatePrg
Selected Cars : Set(Audi, i20, Santro, BMW, Benz, Creta, i10, Swift)
gopal@ubuntu:~/SCALA$
```

Set Operations

```
gopal@ubuntu:~/SCALA$ cat setOperations.scala
object setOperations
 {
   def main(args: Array[String])
   {
      val ranks = Set(5,6,9,20,30,45);
      val ranksnew = Set(1,3,20,2,45,121);

      println("MAX Rank in Ranks : " + ranks.max);
      println("MIN Rank is Ranks : " + ranks.min);

      println( "ranks.intersect(ranksnew) : " + ranks.intersect(ranksnew) );
   }
}
gopal@ubuntu:~/SCALA$
```

```
gopal@ubuntu:~/SCALA$ nano setOperations.scala
gopal@ubuntu:~/SCALA$ scalac setOperations.scala
gopal@ubuntu:~/SCALA$
gopal@ubuntu:~/SCALA$ scala setOperations
MAX Rank in Ranks : 45
MIN Rank is Ranks : 5
ranks.intersect(ranksnew) : Set(20, 45)
gopal@ubuntu:~/SCALA$
```

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.**

➔ **There are two kinds of Maps, the immutable and the mutable**

```
gopal@ubuntu:~/SCALA$ cat mapPrg.scala
object mapPrg
{
   def main(args: Array[String])
   {
      val natureOfTech = Map("Hadoop" -> "BIGDATA",
                             "Spark" -> "IN-MEMORY",
                             "IoT" -> "AI" );
      println( "Keys in natureOfTech : " + natureOfTech.keys )
      println( "Values in natureOfTech : " + natureOfTech.values )
  }
}
gopal@ubuntu:~/SCALA$
```

```
gopal@ubuntu:~/SCALA$ scalac mapPrg.scala
gopal@ubuntu:~/SCALA$
gopal@ubuntu:~/SCALA$ scala mapPrg
Keys in natureOfTech : Set(Hadoop, Spark, IoT)
Values in natureOfTech : MapLike(BIGDATA, IN-MEMORY, AI)
gopal@ubuntu:~/SCALA$
```

```
scala> val nums = Map("one"->1,"two"->2,"three"->3);
nums: scala.collection.immutable.Map[String,Int] = Map(one -> 1, two -> 2, three
 -> 3)

scala> nums.get("two");
res0: Option[Int] = Some(2)

scala> nums.get("three");
res1: Option[Int] = Some(3)

scala>
```

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**

```
scala> val b = new Tuple1(1,"Gopal",'g',123.44);
b: ((Int, String, Char, Double),) = ((1,Gopal,g,123.44),)

scala> val b = (1,"Gopal",'g',123.44);
b: (Int, String, Char, Double) = (1,Gopal,g,123.44)

scala>
```

```
scala> val x = (10,20,30,40);
x: (Int, Int, Int, Int) = (10,20,30,40)

scala> val sum = x._1+x._2+x._3+x._4;
sum: Int = 100

scala>
```

Tuple.productIterator() method to iterate over all the elements of a **Tuple**

```
scala> val x = (10,20,30,40);
x: (Int, Int, Int, Int) = (10,20,30,40)

scala> x.productIterator.foreach{ i =>println("Value = " + i )}
Value = 10
Value = 20
Value = 30
Value = 40

scala>
```

```
scala> val y = ("Gopal","Krishna");
y: (String, String) = (Gopal,Krishna)

scala> println(" Swapped Tuple is --> " + y.swap);
 Swapped Tuple is --> (Krishna,Gopal)

scala>
```

```
scala> var tuple1 = (10,20,30,40)
tuple1: (Int, Int, Int, Int) = (10,20,30,40)

scala> var tuple2 = ("gopal","krishna","niyathi")
tuple2: (String, String, String) = (gopal,krishna,niyathi)

scala> var tuple3 = (100,"GK",'V',true,123.44)
tuple3: (Int, String, Char, Boolean, Double) = (100,GK,V,true,123.44)

scala> tuple1._2
res12: Int = 20

scala> tuple2._1
res13: String = gopal

scala> tuple3._4
res14: Boolean = true
```

Flatten: flatten collapses one level of nested structure

```
scala> List(List(1, 2), List(3, 4)).flatten
res7: List[Int] = List(1, 2, 3, 4)

scala>
```

flatMap: flatMap is a frequently used combinator that combines mapping and flattening. flatMap takes a function that works on the **nested lists** and then concatenates the results back together.

```
scala>  val nestedNumbers = List(List(1, 2), List(3, 4))
nestedNumbers: List[List[Int]] = List(List(1, 2), List(3, 4))

scala> nestedNumbers.flatMap(x => x.map(_ * 2))
res11: List[Int] = List(2, 4, 6, 8)

scala> nestedNumbers.flatMap(x => x.map(_ + 10))
res12: List[Int] = List(11, 12, 13, 14)

scala>
```

Pattern Matching in Scala

A pattern match includes a sequence of alternatives, each starting with the keyword **case**. Each alternative includes a pattern and one or more expressions, which will be evaluated if the pattern matches.

=> Symbol separates the **pattern** from **expressions**

Example:

```
gopal@ubuntu:~/SCALA$ cat casePrg.scala
object casePrg
{
  def main(args: Array[String])
   {
      var days = List("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")
      for (day <- days) {
           println(day)
      day match {
        case "Mon" => println("First Day of the Week")
        case "Tue" => println("Second Day of the Week")
        case "Wed" => println("Third Day of the Week")
        case "Thu" => println("Fourth Day of the Week")
        case "Fri" => println("Fifth Day of the Week")
        case "Sat" => println("Sixth Day of the Week")
        case "Sun" => println("Seventh Day of the Week")
        case _     => println("Not a Day..")
    }

   }
}
}
```

```
gopal@ubuntu:~/SCALA$ scalac casePrg.scala
gopal@ubuntu:~/SCALA$
gopal@ubuntu:~/SCALA$ scala casePrg
Mon
First Day of the Week
Tue
Second Day of the Week
Wed
Third Day of the Week
Thu
Fourth Day of the Week
Fri
Fifth Day of the Week
Sat
Sixth Day of the Week
Sun
Seventh Day of the Week
gopal@ubuntu:~/SCALA$
```

CURRYING FUNCTION IN SCALA

Currying transforms a function that takes **multiple parameters** into a **chain of functions**, each taking a single parameter. Curried functions are defined with multiple parameter lists, as follows:

def strcat(s1: String)(s2: String) = s1 + s2

Alternatively, you can also use the following syntax to define a curried function:

def strcat(s1: String) = (s2: String) => s1 + s2

Following is the syntax to call a curried function:

```
scala> def gopalfunction(s1: String)(s2: String) = s1 + s2

gopalfunction: (s1: String)(s2: String)String
scala> gopalfunction("GOPAL")("KRISHNA")

res7: String = GOPALKRISHNA

scala> gopalfunction("GOPAL")("    KRISHNA")

res8: String = GOPAL    KRISHNA

scala> gopalfunction("GOPAL")("\tKRISHNA")

res9: String = GOPALKRISHNA

scala> gopalfunction("GOPAL")("\nKRISHNA")

res10: String =

GOPAL

KRISHNA

++++++++++++++++++++++++++++++++++++++++++++++
scala> def abcFun(x:Int)(y:Int) = x * y

abcFun: (x: Int)(y: Int)Int

scala> abcFun(4)(5)

res11: Int = 20

scala> abcFun(4)(5)(6)

<console>:10: error: Int does not take parameters

        abcFun(4)(5)(6)
```

```
scala> def abcFun(x:Int)(y:Int)(z:Int) = x * y + z

abcFun: (x: Int)(y: Int)(z: Int)Int

scala> abcFun(2)(3)(4)

res13: Int = 10

scala> def TestFun(x:Int)(y:Int)(z:String) = x * y + z

TestFun: (x: Int)(y: Int)(z: String)String

scala> TestFun(2)(3)("AAA")

res14: String = 6AAA
```

Creating Mutable Collections from Immutable Ones

```
scala> val m = Map("SPARK" -> 597, "SCALA" -> 40)

m: scala.collection.immutable.Map[String,Int] = Map(SPARK -> 597, SCALA -> 40)

scala> m = Map("Java" -> 200)

<console>:8: error: reassignment to val

      m = Map("Java" -> 200)

// To convert the above Immutable Map to Mutable Map

scala> val n = m.toBuffer

n: scala.collection.mutable.Buffer[(String, Int)] = ArrayBuffer((SPARK,597), (SCALA,40))

scala> n += ("java" -> 200)

res0: n.type = ArrayBuffer((SPARK,597), (SCALA,40), (java,200))
```

To Trim of certain Elements:

```
scala> n trimStart 1
scala> println(n)

ArrayBuffer((SCALA,40),  (java,200))

//Below buffer of tuples is now an immutable map again

scala> val x = n.toMap

x: scala.collection.immutable.Map[String,Int] = Map(SCALA -> 40, java -> 200)
```

scala> println(n)

ArrayBuffer((Java,201), (Java,200))

scala> n += ("Java" -> 200)

res7: n.type = ArrayBuffer((Java,201), (Java,200), (Java,200))

scala> n += ("Java" -> 200)

res7: n.type = ArrayBuffer((Java,201), (Java,200), (Java,200))

scala> n += ("SCALA" -> 40)

res8: n.type = ArrayBuffer((Java,201), (Java,200), (Java,200), (SCALA,40))

Examples on Lists & Sets

scala> val l = n.toList

l: List[(String, Int)] = List((Java,201), (Java,200), (Java,200), (SCALA,40))

scala> val s = n.toSet

s: scala.collection.immutable.Set[(String, Int)] = Set((Java,201), (Java,200), (SCALA,40))

NOTE: The list "l" and set "s" were created successfully, with the **list** containing the **duplicated entries** and the **set** restricted to contain only **unique entries**.

3.Spark RDD Operations

i. Create RDDs from different data sources

ii. Perform basic RDD transformations (map, filter, reduce).

iii. Introduce pair RDDs and perform key-based transformations.

SPARK framework is from **Apache Vendor** which is open source

Spark is only ment for **BIGDATA PROCESSING**(**using IN_MEMORY** Cluster Computing Primitive)..However its not ment for STORAGE.

SPARK does not have any storage compenet… and it can READ the InputData from **LFS,HDFS,RDBMS,NOSQL,CLOUD(AZURE/AWS**)

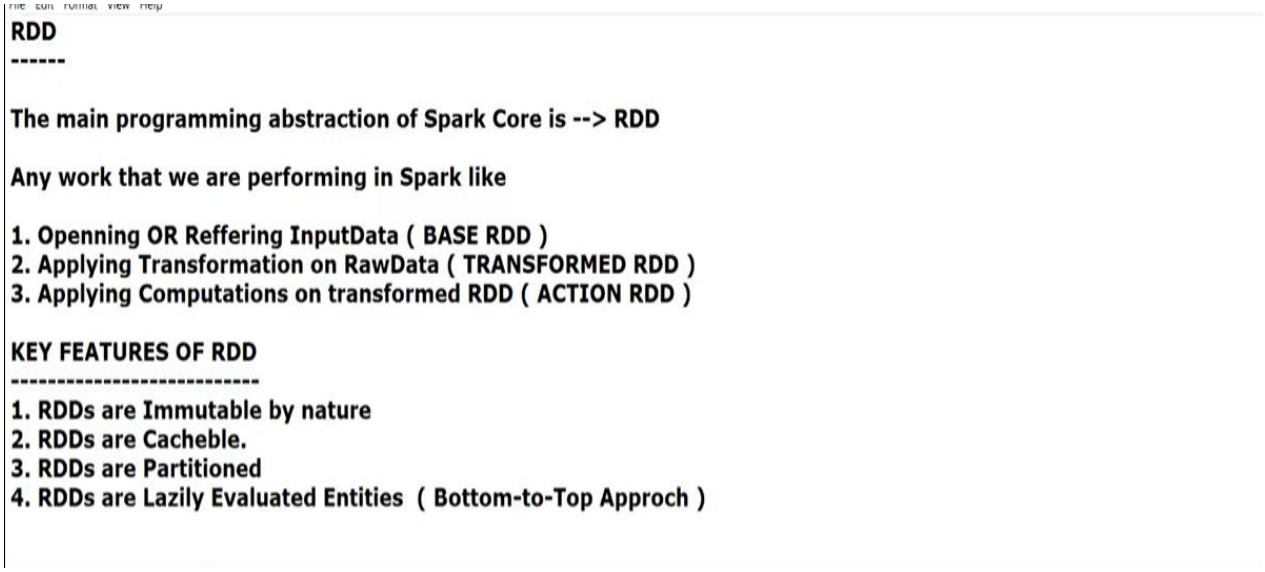
Once the processing is done from Spark ,it can WRITE the output data any of the above storage systems.

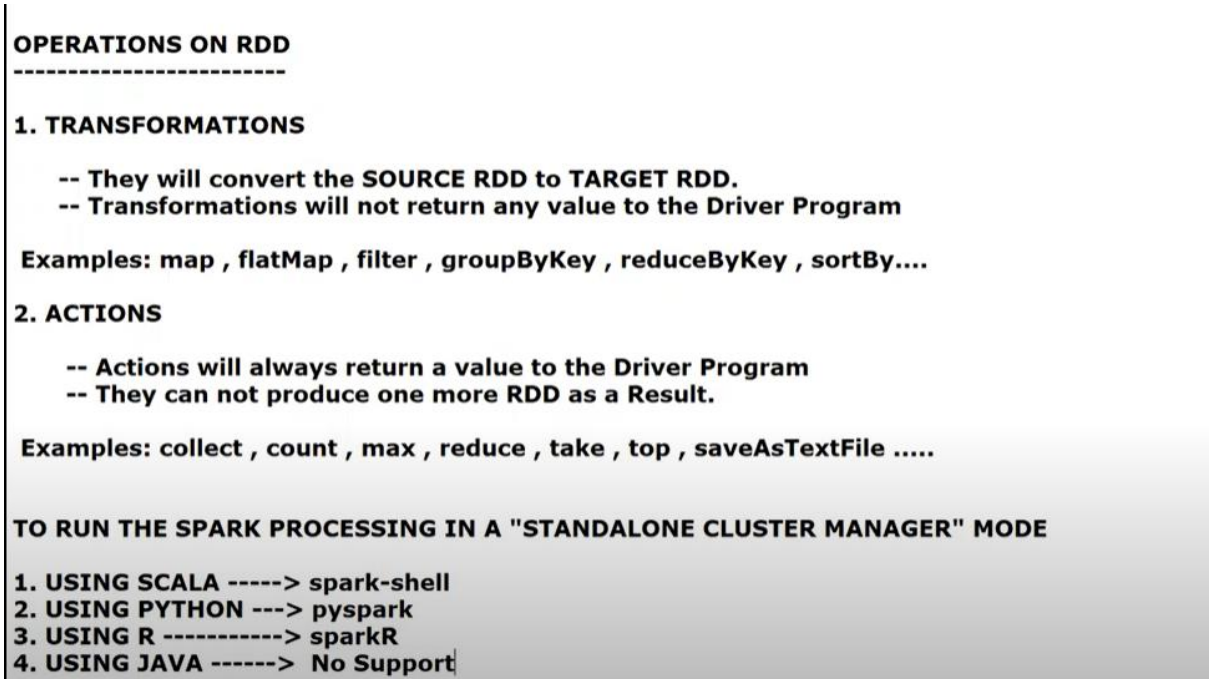
Hadoop(HDFS&MAPREDUCE),HIVE,HBASE,SQOOP,OOZIE

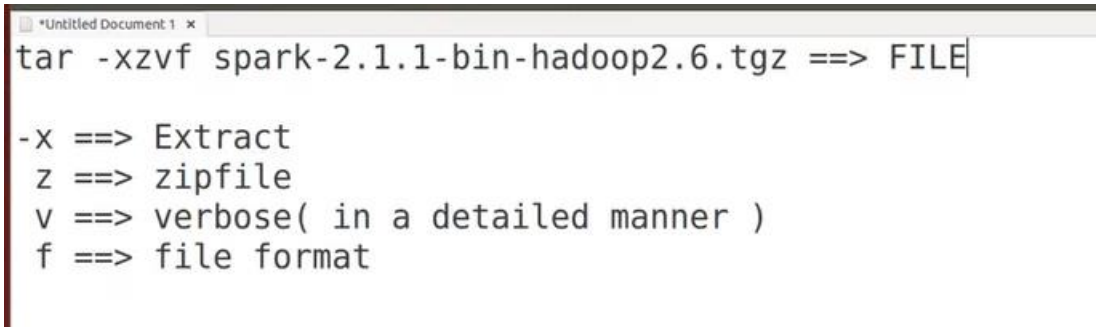
, FLUME…..

SPARK (spark core, spark SQL, spark Streaming, sparkMLLib, spark Graph-X)--→ unified stack(these are 5 different modules)

Spark processing can be achieved through-→scala, python(pyspark), R, java







Resilient Distributed Datasets (RDD)

RDD represents a collection of partitioned data elements that can be operated on in parallel. It is the primary data abstraction mechanism in Spark.

It is defined as an abstract class in the Spark library. Conceptually, RDD is similar to a Scala collection, except that it represents a distributed dataset and it supports lazy operations

RDD OPERATIONS

Spark applications process data using the methods defined in the RDD class or classes derived from it. These methods are also referred to as operations.

RDD operations can be categorized into two types:

 Transformations

 Actions

Transformations

 A transformation method of an RDD creates a new RDD by performing a computation on the source RDD

 RDD transformations are conceptually similar to Scala collection methods.

 The key difference is that the Scala collection methods operate on data that can fit in the memory of a single machine, whereas RDD methods can operate on data distributed across a cluster of nodes

 RDD transformations are lazy, whereas Scala collection methods are strict.

ii. Perform basic RDD transformations (map, filter, reduce).

1.1 Map:

The map method is a higher-order method that takes a function as input and applies it to each element in the source RDD to create a new RDD

gopal@ubuntu:~$ hadoop fs -cat /SparkInputDir/Input.log hadoop is one of the bigdata tool

bigdata is not only for hadoop

hadoop is ment for bigdata storage and processing hadoop is good for analytics also

thanks hadoop gopal@ubuntu:~$

Example Code:

scala> val file = sc.textFile("hdfs://localhost:54310/SparkInputDir/Input.log")
scala> val fileLength = file.map(l => l.length)

scala> fileLength.collect OUTPUT:

res0: Array[Int] = Array(33, 30, 49, 33, 13)

1.2 filter:

 The filter method is a higher-order method that takes a Boolean function as input and applies it to each element in the source RDD to create a new RDD.

 A Boolean function takes an input and returns true or false.

 The filter method returns a new RDD formed by selecting only those elements for which the input Boolean function returned true.

 Thus, the new RDD contains a subset of the elements in the original RDD.

Example Code:

scala> val file = sc.textFile("hdfs://localhost:54310/SparkInputDir/Input.log")
scala> val fileLength = file.filter(l => l.length >= 33)

scala> fileLength.collect

OUTPUT:

Array[String] = Array(hadoop is one of the bigdata tool, hadoop is ment for bigdata storage and processing, hadoop is good for analytics also)

scala> val fileLength = file.filter(l => l.length == 30)

OUTPUT:

Array[String] = Array(bigdata is not only for hadoop) scala> val fileLength = file.filter(l => l.length != 33)

OUTPUT:

Array[String] = Array(bigdata is not only for hadoop, hadoop is ment for bigdata storage and processing, thanks hadoop)

scala> val fileLength = file.filter(l => l.contains("analytics"))

OUTPUT:Array[String] = Array(hadoop is good for analytics also)

reduce:

The higher-order reduce method aggregates the elements of the source RDD using an associative and commutative binary operator provided to it.

Example:

scala> val data = sc.parallelize(List(10,20,10,30,50,70)) scala> val sum = data.reduce((x,y) => (x + y)) OUTPUT:

sum: Int = 190

scala> val multipliedProduct = data.reduce((x,y) => (x * y))

OUTPUT:

multipliedProduct: Int = 210000000

iii. Introduce pair RDDs and perform key-based transformations.

GroupByKey:

- The groupByKey method returns an RDD of pairs, where the first element in a pair is a key from the source RDD and the second element is a collection of all the values that have the same key.
- It is similar to the groupBy method. The difference is that groupBy is a higher-order method that takes as
- input a function that returns a key for each element in the source RDD.
- The groupByKey method operates on an RDD of key-value pairs, so a key generator function is not required as input

Example Code:

scala> val data2 = sc.parallelize(List((10,1),(20,2),(10,3),(10,4),(20,8),(40,9),(20,5),(10,6)))

scala> val groupByKeyData = data2.groupByKey() scala> groupByKeyData.collect

OUTPUT:

Array[(Int, Iterable[Int])] = Array((40,CompactBuffer(9)), (20,CompactBuffer(2, 8, 5)), (10,CompactBuffer(1, 3, 4, 6)))

ReduceByKey:

- The higher-order reduceByKey method takes an associative binary operator as input and reduces values with the same key to a single value using the specified binary operator.
- A binary operator takes two values as input and returns a single value as output.
- An associative operator returns the same result regardless of the grouping of the operands.
- The reduceByKey method can be used for aggregating values by key.
- For example, it can be used for
- Calculating sum,
- Calculating product,

- Calculating minimum or maximum of all the values mapped to the same key.

Example Code :

scala> val iniData = sc.parallelize(List(("Spark",1),("Scala",2),("Spark",3),("Spark",4),("Spark",8),("Java",9),("Scala",5),("Spark", 6)))

scala> val sumOfIniData = iniData.reduceByKey((x,y) => (x+y)) scala> sumOfIniData.collect

OUTPUT:

Array[(String, Int)] = Array((Java,9), (Scala,7), (Spark,22))

scala> val minByKeyData = iniData.reduceByKey((x,y) => if (x < y) x else y) scala> minByKeyData.collect

OUTPUT:

Array[(String, Int)] = Array((Java,9), (Scala,2), (Spark,1))

scala> val maxByKeyData = iniData.reduceByKey((x,y) => if (x > y) x else y) scala> minByKeyData.collect

OUTPUT:

Array[(String, Int)] = Array((Java,9), (Scala,5), (Spark,8))

MapValues:

- The mapValues method is a higher-order method that takes a function as input and applies it to each value in the source RDD.
- It returns an RDD of key-value pairs.
- It is similar to the map method, except that it appliesthe input function only to each value in the source RDD, so the keys are not changed.
- The returned RDD has
- the same keys as the source RDD

Example Code:

scala> val stuNameMarks = sc.parallelize(List(("Raj", 70), ("Ramesh", 89), ("Rakesh", 79)))

scala> val stuMarksFiltered = stuNameMarks.mapValues { x => 2*x} scala> stuMarksFiltered.collect

OUTPUT:

Array[(String, Int)] = Array((Raj,140), (Ramesh,178), (Rakesh,158))

4.Spark DataFrames and SQL

i. Create and manipulate Spark DataFrames using Scala.

ii. Perform common DataFrame operations (select, filter, groupBy).

iii. Write SQL queries with Spark SQL in Scala

iv. Register and query temporary tables.

5.Machine Learning with MLlib

i. Implement a linear regression model using MLlib in Scala

ii. Evaluate the model's performance.

iii. Build a classification model (e.g., Logistic Regression) using MLlib in Scala

iv. Evaluate the classification model.

6.Spark Streaming in Scala

i. Set up a Spark Streaming application using Scala

ii. Process and analyze real-time streaming data

iii. Implement windowed operations on streaming data (e.g., windowed counts).

7.Advanced Spark and Scala Applications

i. Explore the use of broadcast variables and accumulators in Spark.

ii. Create and analyze a graph using Spark's GraphX library.

8. Capstone Project

i. Develop a data science project showcasing data processing, analysis, and machine learning using Spark