**Scala**

**Differentiate between Val and var in Scala.**

Val and var are the two keywords used to define variables in Scala. Var keyword is just similar to variable declaration in Java whereas Val is little different. Once a variable is declared using Val the reference cannot be changed to point to another reference. This functionality of Val keyword in Scala can be related to the functionality of java final keyword. To simplify it, **Val refers to immutable declaration of a variable whereas var refers to mutable declaration of a variable in Scala.**

**Closure in Scala**

Closure is a function in Scala where the return value of the function depends on the value of one or more variables that have been declared outside the function.

var factor = 3

val multiplier = (i:Int) => i \* factor

factor has a reference to a variable outside the function but in the enclosing scope. The function references factor and reads its current value each time.

**Trait in Scala**

A trait encapsulates method and field definitions, which can then be reused by mixing them into classes. Unlike class inheritance, in which each class must inherit from just one superclass, a class can mix in any number of traits.

**Higher order functions**

Higher order function is a function that either takes a function as argument or returns a function. In other words we can say a function which works with function is called higher order function.

object MainObject {

   def main(args: Array[String]) = {

     functionExample(25, multiplyBy2)                   // Passing a function as parameter

    }

    def functionExample(a:Int, f:Int=>AnyVal):Unit = {

        println(f(a))                                   // Calling that function

    }

    def multiplyBy2(a:Int):Int = {

        a\*2

    }

}

**Currying in scala**

In scala, method may have multiple parameter lists. When a method is called with a fewer number of parameter lists, then this will yield a function taking the missing parameter lists as its arguments.

object MainObject {

    def add(a:Int)(b:Int) = {

        a+b

    }

    def main(args: Array[String]) = {

        var result = add(10)(10)

        println("10 + 10 = "+result)

        var addIt = add(10)\_

        var result2 = addIt(3)

        println("10 + 3 = "+result2)

    }

}

**Case Classes**

Scala case classes are just regular classes which are immutable by. It uses equal method to compare instance structurally. It does not use new keyword to instantiate object. All the parameters listed in the case class are public and immutable by default.

case class CaseClass(a:Int, b:Int)

object MainObject{

    def main(args:Array[String]){

        var c =  CaseClass(10,10)       // Creating object of case class

        println("a = "+c.a)               // Accessing elements of case class

        println("b = "+c.b)

    }

}

**Spark**

Apache Spark is a **lightning-fast cluster computing technology**, designed for fast computation. It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing. The main feature of Spark is its **in-memory cluster computing that increases the processing speed of an application.**

**Architecture**

In distributed mode, Spark uses a master/slave architecture with one central coordinator and many distributed workers. The central coordinator is called the **driver**. The driver communicates with a potentially large number of distributed workers called **executors**.

**Spark-context**

Spark context is the entry point of spark functionality. It allows spark driver application to access spark cluster with the help of Resource Manager. To create SparkContext, SparkConf should be created.

**Spark core**

Spark Core contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems, and more.

**RDD (Resilient Distributed Datasets)**

* Immutable distributed collection of fault tolerant objects
* Split into partitions which may be computed on various nodes of a clusters
* Users create RDDs in two ways: by loading an external dataset, or by distributing a collection of objects (e.g., a list or set) in their driver program
* Once loaded 2 operations – transformations and actions
* Transformations – operations on RDDs that return a new RDD. E.g. map() and filter()
* Actions – operations on RDDs that return a result to driver program or write it to storage e.g. count() and first()

**RDD Persistence:**

When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset. Future actions are much faster. Caching is a key tool for iterative algorithms and fast interactive use.

Cache() and persist()

Cache() is a form of persist on RDD with the default storage level of MEMORY\_ONLY. Persist() has other storage levels as below:

MEMORY\_ONLY

MEMORY\_AND\_DISK

MEMORY\_ONLY\_SER

MEMORY\_ONLY\_DISK\_SER

DISK\_SER

MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2 , etc

**Scala word count**

val f = sc.textFile("data/README.md")

val wc = f.flatMap(l => l.split(" ")).map(word => (word, 1)).reduceByKey(\_ + \_)

wc.saveAsTextFile("wc\_out")

**Spark streaming program example**

import org.apache.spark.streaming.\_

import org.apache.spark.streaming.StreamingContext.\_

// create a StreamingContext with a SparkConf configuration

val ssc = new StreamingContext(sparkConf, Seconds(10))

// create a DStream that will connect to serverIP:serverPort

val lines = ssc.socketTextStream(“localhost”, 9999)

// split each line into words

val words = lines.flatMap(\_.split(" "))

// count each word in each batch

val pairs = words.map(word => (word, 1))

val wordCounts = pairs.reduceByKey(\_ + \_)

// print a few of the counts to the console

wordCounts.print()

ssc.start() // start the computation

ssc.awaitTermination() // wait for the computation to terminate