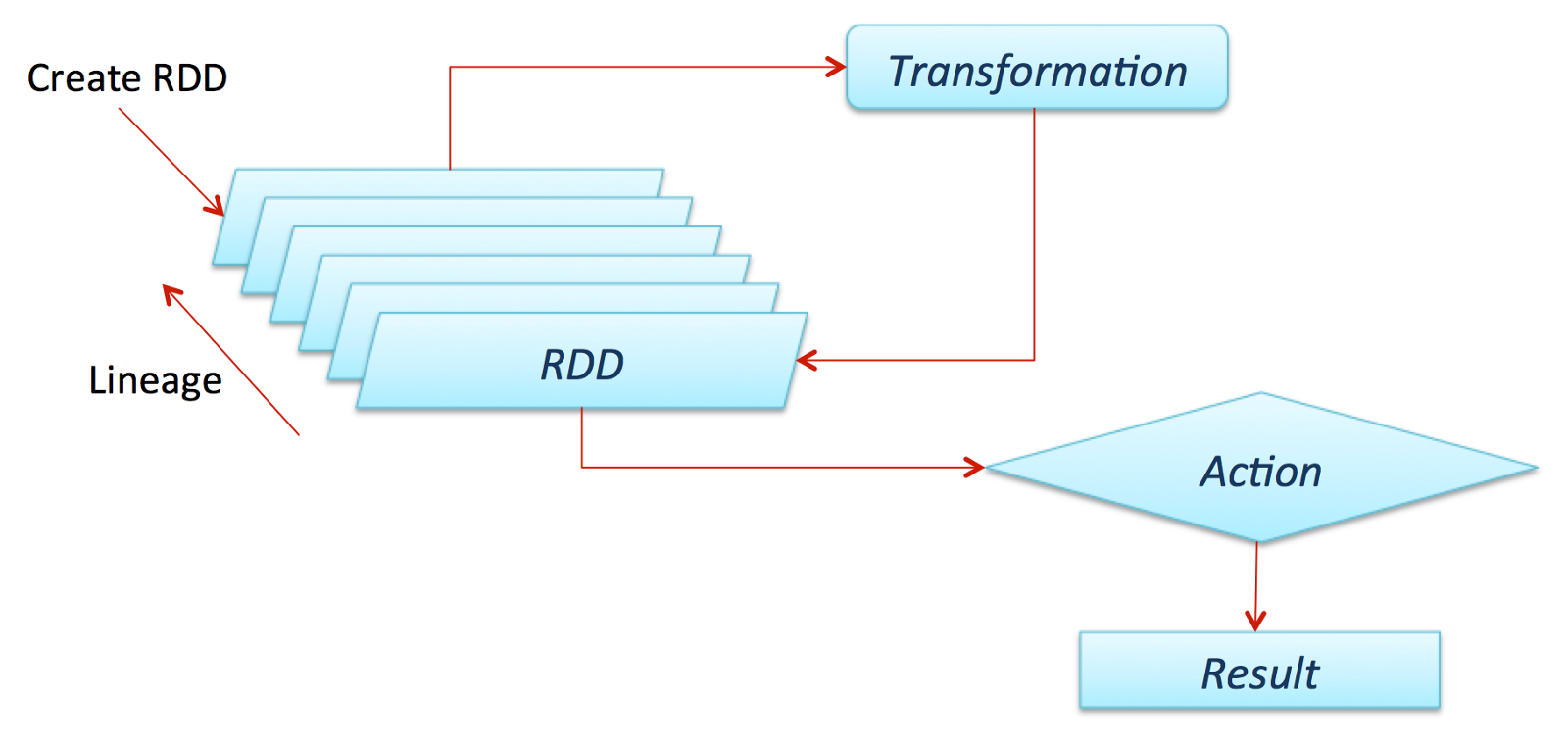
The main abstraction and the beginnings of Apache Spark is the Resilient Distributed Dataset (RDD).

An RDD is a fault-tolerant collection of elements that can be operated on in parallel. You can create them parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.



Something very important to know about Spark is that all **transformations**(we will define it soon) are lazy, that menas that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an **action** requires a result to be returned to the driver program.

By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also *persist* an RDD in memory using the persist (or cache) method, in which case Spark will keep the elements around on the cluster for much faster access the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

Scala Program Example

Let's see the simple program of scala. A detailed description of this program is given in next chapters.

1. **object** MainObject{
2. **def** main(args:Array[String]){
3. print("Hello Scala")
4. }
5. }

Scala String Interpolation Example

This program use string interpolation in print function. You can see the advantage of interpolation. Here, we did not use + operator to concatenate string objects.

1. **class** StringExample{
2. **var** pi = 3.14
3. **def** show(){
4. println(s"value of pi = $pi")
5. }
6. }
8. **object** MainObject{
9. **def** main(args:Array[String]){
10. **var** s = **new** StringExample()
11. s.show()
12. }
13. }

# **Scala Try Catch**

Scala provides try and catch block to handle exception. The try block is used to enclose suspect code. The catch block is used to handle exception occurred in try block. You can have any number of try catch block in your program according to need.

## Scala Try Catch Example

In the following program, we have enclosed our suspect code inside try block. After try block we have used a catch handler to catch exception. If any exception occurs, catch handler will handle it and program will not terminate abnormally.

1. **class** ExceptionExample{
2. **def** divide(a:**Int**, b:**Int**) = {
3. **try**{
4. a/b
5. }**catch**{
6. **case** e: ArithmeticException => println(e)
7. }
8. println("Rest of the code is executing...")
9. }
10. }
11. **object** MainObject{
12. **def** main(args:Array[String]){
13. **var** e = **new** ExceptionExample()
14. e.divide(100,0)
16. }
17. }

Output:

java.lang.ArithmeticException: / by zero

Rest of the code is executing...

## Scala Try Catch Example 2

In this example, we have two cases in our catch handler. First case will handle only arithmetic type exception. Second case has Throwable class which is a super class in exception hierarchy. The second case is able to handle any type of exception in your program. Sometimes when you don't know about the type of exception, you can use super class.

1. **class** ExceptionExample{
2. **def** divide(a:**Int**, b:**Int**) = {
3. **try**{
4. a/b
5. **var** arr = Array(1,2)
6. arr(10)
7. }**catch**{
8. **case** e: ArithmeticException => println(e)
9. **case** ex: Throwable =>println("found a unknown exception"+ ex)
10. }
11. println("Rest of the code is executing...")
12. }
13. }
14. **object** MainObject{
15. **def** main(args:Array[String]){
16. **var** e = **new** ExceptionExample()
17. e.divide(100,10)
19. }
20. }

## Scala Creating a File Example

Scala doesn't provide file writing methods. So, you have to use the Java PrintWriter or FileWriter methods.

1. **import** java.io.\_
2. **val** fileObject = **new** File("ScalaFile.txt" )     // Creating a file
3. **val** printWriter = **new** PrintWriter(fileObject)       // Passing reference of file to the printwriter
4. printWriter.write("Hello, This is scala file")  // Writing to the file
5. printWriter.close()             // Closing printwriter

The above code will create a text file ScalaFile.txt. After creating file printwriter is used to write content to this file.

### **Scala Reading File Example: Reading Each Charecter**

1. **import** scala.io.Source
3. **object** MainObject{
4. **def** main(args:Array[String]){
5. **val** filename = "ScalaFile.txt"
6. **val** fileSource = Source.fromFile(filename)
7. **while**(fileSource.hasNext){
8. println(fileSource.next)
9. }
10. fileSource.close()
11. }
12. }

Output:

H

e

l

l

o

,

T

h

i

s

i

s

s

c

a

l

a

f

i

l

e

Scala provides file reading methods. In the above code, we have imported scala.io package and Source class is used to access file related methods like fromFile().

You must close file refernce after reading or writing file by using close() method.

### **Scala Reading a File Example: Reading Each Line**

1. **import** scala.io.Source
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **val** filename = "ScalaFile.txt"
5. **val** fileSource = Source.fromFile(filename)
6. **for**(line<-fileSource.getLines){
7. println(line)
8. }
9. fileSource.close()
10. }
11. }

# **Scala Tuples**

A tuple is a collection of elements in ordered form. If there is no element present, it is called empty tuple. You can use tuple to store any type of data. You can store similar type or mix type data also. You can use it to return multiple values from a function

## Scala Tuple Example

In the following example, we have created tuple of different types of elements.

1. **object** MainObject{
2. **def** main(args:Array[String]){
3. **var** tuple = (1,5,8,6,4)                     // Tuple of integer values
4. **var** tuple2 = ("Apple","Banana","Gavava")        // Tuple of string values
5. **var** tuple3 = (2.5,8.4,10.50)                // Tuple of float values
6. **var** tuple4 = (1,2.5,"India")                // Tuple of mix type values
7. println(tuple)
8. println(tuple2)
9. println(tuple3)
10. println(tuple4)
11. }
12. }

Output:

(1,5,8,6,4)

(Apple,Banana,Gavava)

(2.5,8.4,10.5)

(1,2.5,India)

### **Scala Tuple Example: Accessing Tuple Elements**

In this example, we are accessing tuple elements by using index. Here, we are using productIterator for iterating tuple elements.

1. **object** MainObject{
2. **def** main(args:Array[String]){
3. **var** tupleValues = (1,2.5,"India")
4. println("iterating values: ")
5. tupleValues.productIterator.foreach(println)    // Iterating tuple values using productIterator
6. println("Accessing values: ")
7. println(tupleValues.\_1) // Fetching first value
8. println(tupleValues.\_2) // Fetching second value
9. }
10. }

Output:

iterating values:

1

2.5

India

Accessing values:

1

2.5

# **Scala Variables and Data Types**

Variable is a name which is used to refer memory location. You can create mutable and immutable variable in scala. Let's see how to declare variable.

## Mutable Variable

You can create mutable variable using **var** keyword. It allows you to change value after declaration of variable.

1. **var** data = 100
2. data = 101  // It works, No error.

In the above code, **var** is a keyword and data is a variable name. It contains an integer value 100. Scala is a type infers language so you don?t need to specify data type explicitly. You can also mention data type of variable explicitly as we have used in below.

### **Another example of variable**

1. **val** data:**Int** = 100  // Here, we have mentioned Int followed by : (colon)

## Immutable Variable

1. **val** data = 100
2. data = 101  // Error: reassignment to val

The above code throws an error because we have changed content of immutable variable, which is not allowed. So if you want to change content then it is advisable to use **var** instead of **val**.

## Data Types in Scala

Data types in scala are much similar to java in terms of their storage, length, except that in scala there is no concept of primitive data types every type is an object and starts with capital letter. A table of data types is given below. You will see their uses further.

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Default Value** | **Size** |
| Boolean | False | True or false |
| Byte | 0 | 8 bit signed value (-27 to 27-1) |
| Short | 0 | 16 bit signed value(-215 to 215-1) |
| Char | '\u0000' | 16 bit unsigned Unicode character(0 to 216-1) |
| Int | 0 | 32 bit signed value(-231 to 231-1) |
| Long | 0L | 64 bit signed value(-263 to 263-1) |
| Float | 0.0F | 32 bit IEEE 754 single-precision float |
| Double | 0.0D | 64 bit IEEE 754 double-precision float |
| String | Null | A sequence of characters |

Scala If-Else-If Ladder Example

1. **var** number:**Int** = 85
2. **if**(number>=0 && number<50){
3. println ("fail")
4. }
5. **else** **if**(number>=50 && number<60){
6. println("D Grade")
7. }
8. **else** **if**(number>=60 && number<70){
9. println("C Grade")
10. }
11. **else** **if**(number>=70 && number<80){
12. println("B Grade")
13. }
14. **else** **if**(number>=80 && number<90){
15. println("A Grade")
16. }
17. **else** **if**(number>=90 && number<=100){
18. println("A+ Grade")
19. }
20. **else** println ("Invalid")

Scala If Statement as better alternative of Ternary Operators

In scala, you can assign if statement result to a function. Scala does not have ternary operator concept like C/C++ but provides more powerful *if* which can return value. Let's see an example

**Example**

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **val** result = checkIt(-10)
4. println (result)
5. }
6. **def** checkIt (a:**Int**)  =  **if** (a >= 0) 1 **else** -1    // Passing a if expression value to function
7. }

Scala Pattern Matching Example

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** a = 1
4. a **match**{
5. **case** 1 => println("One")
6. **case** 2 => println("Two")
7. **case** \_ => println("No")
8. }
9. }
10. }

Scala Pattern Matching Example2

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** result = search ("Hello")
4. print(result)
5. }
6. **def** search (a:Any):Any = a **match**{
7. **case** 1  => println("One")
8. **case** "Two" => println("Two")
9. **case** "Hello" => println("Hello")
10. **case** \_ => println("No")
12. }
13. }

Scala while loop Example

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** a = 10;                       // Initialization
4. **while**( a<=20 ){                // Condition
5. println(a);
6. a = a+2                        // Incrementation
7. }
8. }
9. }

Scala for-loop example by using *to* keyword

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

Scala Break Example

1. **import** scala.util.control.Breaks.\_                  // Importing  package
2. **object** MainObject {
3. **def** main(args: Array[String]) {
4. breakable {                                 // Breakable method to avoid exception
5. **for**(i<-1 **to** 10 by 2){
6. **if**(i==7)
7. break                           // Break used here
8. **else**
9. println(i)
10. }
11. }
12. }
13. }

# **Scala Comments**

The scala comments are statements which are not executed by the compiler or interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code details.

In scala, there are three types of comments

1. Single line comment
2. Multiline comment
3. Documentation comment

## Scala Single Line Comment Example

Single line comment is used to comment single line of code.

1. // Example of single line comment.
2. **object** MainObject {
3. **def** main(args: Array[String]) {
4. **var** a = 1           // Here, a is a variable
5. println(a)
6. }
7. }

Scala Multiline Comment

Multiline comment is used to comment multiple lines of code in the program.

1. // Example of multi line comment.
2. **object** MainObject {
3. **def** main(args: Array[String]) {
4. **var** a = 1
5. println(a)
6. }
7. /\*
8. In the main method, we have created a variable named a
9. and printed it
10. \*/
11. }

Scala Function Declaration Syntax

1. **def** functionName(parameters : typeofparameters) : returntypeoffunction = {
2. // statements to be executed
3. }

In the above syntax, = (equal) operator is looking strange but don't worry scala has defined it as:

You can create function with or without = (equal) operator. If you use it, function will return value. If you don't use it, your function will not return anything and will work like subroutine.

Scala functions don?t use return statement. Return type infers by compiler from the last expression or statement present in the function.

Scala Function Example without using = Operator

The function defined below is also known as non parameterized function.

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. functionExample()           // Calling function
4. }
5. **def** functionExample()  {        // Defining a function
6. println("This is a simple function")
7. }
8. }

Output:

This is a simple function

Scala Function Example with = Operator

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** result = functionExample()          // Calling function
4. println(result)
5. }
6. **def** functionExample() = {       // Defining a function
7. **var** a = 10
8. a
9. }
10. }

Output:

10

Scala Parameterized Function Example

when using parameterized function you must mention type of parameters explicitly otherwise compiler throws an error and your code fails to compile.

1. **object** MainObject {
2. **def** main(args: Array[String]) = {
3. functionExample(10,20)
4. }
5. **def** functionExample(a:**Int**, b:**Int**) = {
6. **var** c = a+b
7. println(c)
8. }
9. }

Output:

30

Scala Recursion Function

In the program given below, we are multiplying two numbers by using recursive function.

In scala, you can create recursive functions also. Be careful while using recursive function. There must be a base condition to terminate program safely.

1. **object** MainObject {
2. **def** main(args: Array[String]) = {
3. **var** result = functionExample(15,2)
4. println(result)
5. }
6. **def** functionExample(a:**Int**, b:**Int**):**Int** = {
7. **if**(b == 0)          // Base condition
8. 0
9. **else**
10. a+functionExample(a,b-1)
11. }
12. }

Output:

30

Function Parameter with Default Value

Scala provides a feature to assign default values to function parameters. It helps in the scenario when you don't pass value during function calling. It uses default values of parameters.

Let's see an example.

Scala Function Parameter example with default value

1. **object** MainObject {
2. **def** main(args: Array[String]) = {
3. **var** result1 = functionExample(15,2)     // Calling with two values
4. **var** result2 = functionExample(15)   // Calling with one value
5. **var** result3 = functionExample()     // Calling without any value
6. println(result1+"\n"+result2+"\n"+result3)
7. }
8. **def** functionExample(a:**Int** = 0, b:**Int** = 0):**Int** = {   // Parameters with default values as 0
9. a+b
10. }
11. }

Output:

17

15

0

Scala Function Named Parameter Example

In scala function, you can specify the names of parameters during calling the function. In the given example, you can notice that parameter names are passing during calling. You can pass named parameters in any order and can also pass values only.

Let's see an example.

1. **object** MainObject {
2. **def** main(args: Array[String]) = {
3. **var** result1 = functionExample(a = 15, b = 2)    // Parameters names are passed during call
4. **var** result2 = functionExample(b = 15, a = 2)    // Parameters order have changed during call
5. **var** result3 = functionExample(15,2)             // Only values are passed during call
6. println(result1+"\n"+result2+"\n"+result3)
7. }
8. **def** functionExample(a:**Int**, b:**Int**):**Int** = {
9. a+b
10. }
11. }

# **Scala 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.

Higher order function allows you to create function composition, lambda function or anonymous function etc.

Let's see an example.

## Scala Example: Passing a Function as Parameter in a Function

1. **object** MainObject {
2. **def** main(args: Array[String]) = {
3. functionExample(25, multiplyBy2)                   // Passing a function as parameter
4. }
5. **def** functionExample(a:**Int**, f:**Int**=>AnyVal):Unit = {
6. println(f(a))                                   // Calling that function
7. }
8. **def** multiplyBy2(a:**Int**):**Int** = {
9. a\*2
10. }
11. }

Output:

60

## Scala Example: Function Composition

In scala, functions can be composed from other functions. It is a process of composing in which a function represents the application of two composed functions.

Let's see an example.

1. **object** MainObject {
2. **def** main(args: Array[String]) = {
3. **var** result = multiplyBy2(add2(10))      // Function composition
4. println(result)
5. }
6. **def** add2(a:**Int**):**Int** = {
7. a+2
8. }
10. **def** multiplyBy2(a:**Int**):**Int** = {
11. a\*2
12. }
13. }

Output:

24

## Scala Anonymous (lambda) Function

Anonymous function is a function that has no name but works as a function. It is good to create an anonymous function when you don't want to reuse it latter.

You can create anonymous function either by using => (rocket) or \_ (underscore) wild card in scala.

Let's see an example.

## Scala Anonymous function Example

1. **object** MainObject {
2. **def** main(args: Array[String]) = {
3. **var** result1 = (a:**Int**, b:**Int**) => a+b        // Anonymous function by using => (rocket)
4. **var** result2 = (\_:**Int**)+(\_:**Int**)              // Anonymous function by using \_ (underscore) wild card
5. println(result1(10,10))
6. println(result2(10,10))
7. }
8. }

Output:

20

20

## Scala Multiline Expression

Expressions those are written in multiple lines are called multiline expression. In scala, be carefull while using multiline expressions.

The following program explains about if we break an expression into multiline, the scala compiler throw a warning message.

## Scala Multiline Expression Example

1. **def** add1(a:**Int**, b:**Int**) = {
2. a
3. +b
4. }

The above program does not evaluate complete expression and just return b here. So, be careful while using multiline expressions.

Output:

MainObject.scala:3: warning: a pure expression does nothing in statement

position; you may be omitting necessary parentheses

a

^

one warning found

10

You can apply following ways to avoid above problem.

## Scala Example Multiline Expression

1. **object** MainObject {
2. **def** add2(a:**Int**, b:**Int**) = {
3. a+
4. b
5. }
6. **def** add3(a:**Int**, b:**Int**) = {
7. (a
8. +b)
9. }
10. **def** main(args: Array[String]) = {
11. **var** result2 = add2(10,10)
12. **var** result3 = add3(10,10)
13. println(result2+"\n"+result3)
14. }
15. }

Output:

20

20

## Scala Function Currying

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.

In other words it is a technique of transforming a function that takes multiple arguments into a function that takes a single argument.

## Scala Function Currying Example

1. **object** MainObject {
2. **def** add(a:**Int**)(b:**Int**) = {
3. a+b
4. }
5. **def** main(args: Array[String]) = {
6. **var** result = add(10)(10)
7. println("10 + 10 = "+result)
8. **var** addIt = add(10)\_
9. **var** result2 = addIt(3)
10. println("10 + 3 = "+result2)
11. }
12. }

Output:

20

## Scala Nested Functions

Scala is a first class function language which means it allows you to passing function, returning function, composing function, nested function etc. An example below explain about how to define and call nested functions.

## Scala Nested Functions Example

1. **object** MainObject {
2. **def** add(a:**Int**, b:**Int**, c:**Int**) = {
3. **def** add2(x:**Int**,y:**Int**) = {
4. x+y
5. }
6. add2(a,add2(b,c))
7. }
8. **def** main(args: Array[String]) = {
9. **var** result = add(10,10,10)
10. println(result)
11. }
12. }

Output:

30

## Scala Function with Variable Length Parameters

In scala, you can define function of variable length parameters. It allows you to pass any number of arguments at the time of calling the function.

Let's see an example.

## Scala Example: Function with Variable Length Parameters

1. **def** add(args: **Int**\*) = {
2. **var** sum = 0;
3. **for**(a <- args) sum+=a
4. sum
5. }
6. **var** sum = add(1,2,3,4,5,6,7,8,9);
7. println(sum);

Scala String equals() Method Example

You can also use equal() method to compare two string objects. It returns true if both string object are equal otherwise returns false.

1. **class** StringExample{
2. **var** s1 = "Scala string example"
3. **var** s2 = "Hello Scala"
4. **var** s3 = "Hello Scala"
5. **def** show(){
6. println(s1.equals(s2))
7. println(s2.equals(s3))
8. }
9. }
11. **object** MainObject{
12. **def** main(args:Array[String]){
13. **var** s = **new** StringExample()
14. s.show()
15. }
16. }

Output:

false

true

Scala compareTo() Method Example

The compareTo() method compares the given string with current string lexicographically. It returns positive number, negative number or 0.

If first string is greater than second string, it returns positive number (difference of character value). If first string is less than second string, it returns negative number and if first string is equal to second string, it returns 0.

1. s1 **>** s2 =**>** positive number
2. s1 **<** s2**s2** =**>** negative number
3. s1 == s2 =**>** 0
4. **class** StringExample{
5. **var** s1 = "Scala string example"
6. **var** s2 = "Hello Scala"
7. **var** s3 = "Hello Scala"
8. **def** show(){
9. println(s1.compareTo(s2))
10. println(s2.compareTo(s3))
11. }
12. }
14. **object** MainObject{
15. **def** main(args:Array[String]){
16. **var** s = **new** StringExample()
17. s.show()
18. }
19. }

Output:

11

0

Scala Concatenation Example by using + (Plus) Operator

In scala, you can concatenate string by using + operator. An example is given below in which two strings are concatenated by using + operator.

1. **class** StringExample{
2. **var** s1 = "This is "+"Scala string example"
3. **def** show(){
4. println(s1)
5. }
6. }
8. **object** MainObject{
9. **def** main(args:Array[String]){
10. **var** s = **new** StringExample()
11. s.show()
12. }
13. }

Output:

This is Scala string example

Scala String concat() Method Example

Apart from + operator, you can also use concat() method to concatenate two strings. It is a predefined method of string class.

1. **class** StringExample{
2. **var** s1 = "This is "
3. **var** s2 = "Scala string example"
4. **def** show(){
5. println(s1.concat(s2))
6. }
7. }
9. **object** MainObject{
10. **def** main(args:Array[String]){
11. **var** s = **new** StringExample()
12. s.show()
13. }
14. }

Output:

This is Scala string example

Scala substring() Method Example

The substring() method is used to get substring from a string. By specifying start and end index as argument you can get substring according to your requirement. It is a predefined method of string class.

1. **class** StringExample3{
2. **var** s1 = "Scala string example"
3. **def** show(){
4. println(s1.substring(0,5))
5. }
6. }
8. **object** MainObject{
9. **def** main(args:Array[String]){
10. **var** s = **new** StringExample3()
11. s.show()
12. }
13. }

Output:

Scala

# **Scala Tuples**

A tuple is a collection of elements in ordered form. If there is no element present, it is called empty tuple. You can use tuple to store any type of data. You can store similar type or mix type data also. You can use it to return multiple values from a function

## Scala Tuple Example

In the following example, we have created tuple of different types of elements.

1. **object** MainObject{
2. **def** main(args:Array[String]){
3. **var** tuple = (1,5,8,6,4)                     // Tuple of integer values
4. **var** tuple2 = ("Apple","Banana","Gavava")        // Tuple of string values
5. **var** tuple3 = (2.5,8.4,10.50)                // Tuple of float values
6. **var** tuple4 = (1,2.5,"India")                // Tuple of mix type values
7. println(tuple)
8. println(tuple2)
9. println(tuple3)
10. println(tuple4)
11. }
12. }

Output:

(1,5,8,6,4)

(Apple,Banana,Gavava)

(2.5,8.4,10.5)

(1,2.5,India)

### **Scala Tuple Example: Accessing Tuple Elements**

In this example, we are accessing tuple elements by using index. Here, we are using productIterator for iterating tuple elements.

1. **object** MainObject{
2. **def** main(args:Array[String]){
3. **var** tupleValues = (1,2.5,"India")
4. println("iterating values: ")
5. tupleValues.productIterator.foreach(println)    // Iterating tuple values using productIterator
6. println("Accessing values: ")
7. println(tupleValues.\_1) // Fetching first value
8. println(tupleValues.\_2) // Fetching second value
9. }
10. }

Output:

iterating values:

1

2.5

India

Accessing values:

1

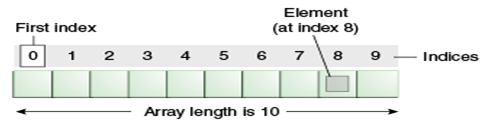
2.5

# **Scala Array**

Array is a collection of mutable values. It is an index based data structure which starts from 0 index to n-1 where n is length of array.

Scala arrays can be generic. It means, you can have an Array[T], where T is a type parameter or abstract type. Scala arrays are compatible with Scala sequences - you can pass an Array[T] where a Seq[T] is required. It also supports all the sequence operations.

Following image represents the structure of array where first index is 0, last index is 9 and array length is 10.



### **Scala Types of array**

1. Single dimensional array
2. Multidimensional array

## Scala Single Dimensional Array

Single dimensional array is used to store elements in linear order. Array elements are stored in contiguous memory space. So, if you have any index of an array, you can easily traverse all the elements of the array.

#### **Syntax for Single Dimensional Array**

1. **var** arrayName : Array[arrayType] = **new** Array[arrayType](arraySize);   or
2. **var** arrayName = **new** Array[arrayType](arraySize)  or
3. **var** arrayName : Array[arrayType] = **new** Array(arraySize);   or
4. **var** arrayName = Array(element1, element2 ? elementN)

## Scala Array Example: Single Dimensional

1. **class** ArrayExample{
2. **var** arr = Array(1,2,3,4,5)      // Creating single dimensional array
3. **def** show(){
4. **for**(a<-arr)                       // Traversing array elements
5. println(a)
6. println("Third Element  = "+ arr(2))        // Accessing elements by using index
7. }
8. }
10. **object** MainObject{
11. **def** main(args:Array[String]){
12. **var** a = **new** ArrayExample()
13. a.show()
14. }
15. }

Output:

1

2

3

4

5

Third Element = 3

## Scala Example 2: Single Dimensional

In this example, we have created an array by using new keyword which is used to initialize memory for array. The entire array elements are set to default value, you can assign that later in your code.

1. **class** ArrayExample{
2. **var** arr = **new** Array[**Int**](5)         // Creating single dimensional array
3. **def** show(){
4. **for**(a<-arr){                      // Traversing array elements
5. println(a)
6. }
7. println("Third Element before assignment = "+ arr(2))        // Accessing elements by using index
8. arr(2) = 10                                                          // Assigning new element at 2 index
9. println("Third Element after assignment = "+ arr(2))
10. }
11. }
13. **object** MainObject{
14. **def** main(args:Array[String]){
15. **var** a = **new** ArrayExample()
16. a.show()
17. }
18. }

Output:

0

0

0

0

0

Third Element before assignment = 0

Third Element after assignment = 10

## Scala Passing Array into Function

You can pass array as an argument to function during function call. Following example illustrate the process how we can pass an array to the function.

1. **class** ArrayExample{
2. **def** show(arr:Array[**Int**]){
3. **for**(a<-arr)                // Traversing array elements
4. println(a)
5. println("Third Element = "+ arr(2))        // Accessing elements by using index
6. }
7. }
9. **object** MainObject{
10. **def** main(args:Array[String]){
11. **var** arr = Array(1,2,3,4,5,6)    // creating single dimensional array
12. **var** a = **new** ArrayExample()
13. a.show(arr)                     // passing array as an argument in the function
14. }
15. }

Output:

1

2

3

4

5

6

Third Element = 3

## Scala Array Example: Iterating By using Foreach Loop

You can also iterate array elements by using foreach loop. Let's see an example.

1. **class** ArrayExample{
2. **var** arr = Array(1,2,3,4,5)      // Creating single dimensional array
3. arr.foreach((element:**Int**)=>println(element))       // Iterating by using foreach loop
4. }
6. **object** MainObject{
7. **def** main(args:Array[String]){
8. **new** ArrayExample()
9. }
10. }

Connect to SQL

import java.sql.{Connection,DriverManager}

object ScalaJdbcConnectSelect extends App {

// connect to the database named "mysql" on port 8889 of localhost

val url = "jdbc:mysql://localhost:8889/mysql"

val driver = "com.mysql.jdbc.Driver"

val username = "root"

val password = "root"

var connection:Connection = \_

try {

Class.forName(driver)

connection = DriverManager.getConnection(url, username, password)

val statement = connection.createStatement

val rs = statement.executeQuery("SELECT host, user FROM user")

while (rs.next) {

val host = rs.getString("host")

val user = rs.getString("user")

println("host = %s, user = %s".format(host,user))

}

} catch {

case e: Exception => e.printStackTrace

}

connection.close

}

Connect to HDFS

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import java.io.PrintWriter;

/\*\*

\* @author ${user.name}

\*/

object App {

//def foo(x : Array[String]) = x.foldLeft("")((a,b) => a + b)

def main(args : Array[String]) {

println( "Trying to write to HDFS..." )

val conf = new Configuration()

//conf.set("fs.defaultFS", "hdfs://quickstart.cloudera:8020")

conf.set("fs.defaultFS", "hdfs://192.168.30.147:8020")

val fs= FileSystem.get(conf)

val output = fs.create(new Path("/tmp/mySample.txt"))

val writer = new PrintWriter(output)

try {

writer.write("this is a test")

writer.write("\n")

}

finally {

writer.close()

println("Closed!")

}

println("Done!")

}

}

conf.set("fs.defaultFS", "hdfs://quickstart.cloudera:8020")

and pass the relative path, like so:

fs.create(new Path("/tmp/mySample.txt"))

to write to the file, call 'write' directly on the output stream returned by fs.create, like so:

val os = fs.create(new Path("/tmp/mySample.txt"))

os.write("This is a test".getBytes)

POM.XML

1. <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">
2. <modelVersion>4.0.0</modelVersion>
3. <groupId>org.glassfish.samples</groupId>
4. <artifactId>scala-helloworld</artifactId>
5. <version>1.0-SNAPSHOT</version>
6. <name>${project.artifactId}</name>
7. <description>My wonderfull scala app</description>
8. <inceptionYear>2010</inceptionYear>
9. <licenses>
10. <license>
11. <name>My License</name>
12. <url>http://....</url>
13. <distribution>repo</distribution>
14. </license>
15. </licenses>
17. <properties>
18. <maven.compiler.source>1.5</maven.compiler.source>
19. <maven.compiler.target>1.5</maven.compiler.target>
20. <encoding>UTF-8</encoding>
21. <scala.version>2.8.0</scala.version>
22. <hadoop-hdfs.version>2.5.0-cdh5.2.0</hadoop-hdfs.version>
23. <hadoop-common.version>2.5.0-cdh5.2.0</hadoop-common.version>
24. </properties>

27. <repositories>
28. <!--
29. <repository>
30. <id>scala-tools.org</id>
31. <name>Scala-Tools Maven2 Repository</name>
32. <url>http://scala-tools.org/repo-releases</url>
33. </repository>
34. -->
35. <repository>
36. <id>Akka repository</id>
37. <url>http://repo.akka.io/releases</url>
38. </repository>
40. <repository>
41. <id>cloudera-repos</id>
42. <name>Cloudera Repos</name>
43. <url>https://repository.cloudera.com/artifactory/cloudera-repos/</url>
44. </repository>
46. </repositories>
48. <!--
49. <pluginRepositories>
50. <pluginRepository>
51. <id>scala-tools.org</id>
52. <name>Scala-Tools Maven2 Repository</name>
53. <url>http://scala-tools.org/repo-releases</url>
54. </pluginRepository>
55. </pluginRepositories>
56. -->
57. <dependencies>
58. <dependency>
59. <groupId>org.scala-lang</groupId>
60. <artifactId>scala-library</artifactId>
61. <version>${scala.version}</version>
62. </dependency>
64. <!-- Test -->
65. <dependency>
66. <groupId>junit</groupId>
67. <artifactId>junit</artifactId>
68. <version>4.8.1</version>
69. <scope>test</scope>
70. </dependency>
71. <dependency>
72. <groupId>org.scala-tools.testing</groupId>
73. <artifactId>specs\_${scala.version}</artifactId>
74. <version>1.6.5</version>
75. <scope>test</scope>
76. </dependency>
77. <dependency>
78. <groupId>org.scalatest</groupId>
79. <artifactId>scalatest</artifactId>
80. <version>1.2</version>
81. <scope>test</scope>
82. </dependency>
83. <!-- necessary to write within HDFS -->
84. <dependency>
85. <groupId>org.apache.hadoop</groupId>
86. <artifactId>hadoop-hdfs</artifactId>
87. <version>${hadoop-hdfs.version}</version>
88. </dependency>
90. <dependency>
91. <groupId>org.apache.hadoop</groupId>
92. <artifactId>hadoop-common</artifactId>
93. <version>${hadoop-common.version}</version>
94. </dependency>
95. </dependencies>
97. <build>
98. <sourceDirectory>src/main/scala</sourceDirectory>
99. <testSourceDirectory>src/test/scala</testSourceDirectory>
100. <plugins>
101. <plugin>
102. <groupId>org.scala-tools</groupId>
103. <artifactId>maven-scala-plugin</artifactId>
104. <version>2.15.0</version>
105. <executions>
106. <execution>
107. <goals>
108. <goal>compile</goal>
109. <goal>testCompile</goal>
110. </goals>
111. <configuration>
112. <args>
113. <arg>-make:transitive</arg>
114. <arg>-dependencyfile</arg>
115. <arg>${project.build.directory}/.scala\_dependencies</arg>
116. </args>
117. </configuration>
118. </execution>
119. </executions>
120. <configuration>
121. <launchers>
122. <launcher>
123. <id>sample</id>
124. <mainClass>org.glassfish.samples.App</mainClass>
125. <args>
126. <arg>${basedir}</arg>
127. </args>
128. </launcher>
129. </launchers>
130. </configuration>
132. </plugin>
133. <plugin>
134. <groupId>org.apache.maven.plugins</groupId>
135. <artifactId>maven-surefire-plugin</artifactId>
136. <version>2.6</version>
137. <configuration>
138. <useFile>false</useFile>
139. <disableXmlReport>true</disableXmlReport>
140. <!-- If you have classpath issue like NoDefClassError,... -->
141. <!-- useManifestOnlyJar>false</useManifestOnlyJar -->
142. <includes>
143. <include>\*\*/\*Test.\*</include>
144. <include>\*\*/\*Suite.\*</include>
145. </includes>
146. </configuration>
147. </plugin>
148. </plugins>
149. </build>
150. </project>