**Scala Programing Concepts**

* Data Types in Scala
* Variable Declaration Syntax
* Condition in Scala
* Loop in Scala
* Comments in Scala
* Function and its Type
* Class, Object
* String Function
* Array
* File Handling

**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 |

# Variable Declaration Syntax

**var** number:**Int** = 101

**var** number:**String** = “Name..”

# Scala Conditional Expressions

Scala provides if statement to test the conditional expressions. It tests boolean conditional expression which can be either true or false. Scala use various types of if else statements.

* If statement
* If-else statement
* Nested if-else statement
* If-else-if ladder statement

## Scala if statement

The scala if statement is used to test condition in scala. If block executes only when condition is true otherwise execution of if block is skipped.

**Syntax**

1. **if**(condition){
2. // Statements to be executed
3. }

## Scala Example: If Statement

1. **var** age:**Int** = 20;
2. **if**(age > 18){
3. println ("Age is greate than 18")
4. }

Output:

Age is greate than 18

## Scala If-Else Statement

The scala if-else statement tests the condition. If the condition is true, if block executes otherwise else block executes.

**Syntax**

1. **if**(condition){
2. // If block statements to be executed
3. } **else** {
4. // Else bock statements to be executed
5. }

## Scala if-else example

1. **var** number:**Int** = 21
2. **if**(number%2==0){
3. println("Even number")
4. }**else**{
5. println("Odd number")
6. }

Output:

Odd number

## Scala If-Else-If Ladder Statement

The scala if-else-if ladder executes one condition among the multiple conditional statements.

**Syntax**

1. **if** (condition1){
2. //Code to be executed if condition1 is true
3. } **else** **if** (condition2){
4. //Code to be executed if condition2 is true
5. } **else** **if** (condition3){
6. //Code to be executed if condition3 is true
7. }
8. ...
9. **else** {
10. //Code to be executed if all the conditions are false
11. }

## 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")

Output:

A Grade

## 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. }

**In the above example, we have implemented a pattern matching.**

Here, match using a variable named *a*. This variable matches with best available case and prints output. Underscore (\_) is used in the last case for making it default case.

Output:

One

Match expression can return case value also. In next example, we are defining method having a match with cases for any type of data. Any is a class in scala which is a super class of all data types and deals with all type of data. Let's see an example.

## 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. }

Def a(){

}

Def a()= if() else …

## Scala Loop

## 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. }

Output:

10

12

14

16

18

20

## Scala Infinite While Loop Example

You can also create an infinite while loop. In the below program, we just passed *true* in while loop. Be careful, while using infinite loop.

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

Output:

10

12

14

16

...

Ctr+Z // To stop execution

## Scala do-while loop example

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

Output:

10

12

14

16

18

20

## Scala Infinite do-while loop

In scala, you can create infinite do-while loop. To create infinite loop just pass *true* literal in loop condition.

Let's see an example.

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

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

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

## Scala for-loop filtering Example

You can use *for* to filter your data. In the below example, we are filtering our data by passing a conditional expression. This program prints only even values in the given range.

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **for**( a <- 1 **to** 10 **if** a%2==0 ){
4. println(a);
5. }
6. }
7. }

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

In the above example, we have used *yield* keyword which returns a result after completing of loop iterations. The for use buffer internally to store iterated result and after finishing all iterations it yields the final result from that buffer. It does not work like imperative loop.

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** result = **for**( a <- 1 **to** 10) yield a
4. **for**(i<-result){
5. println(i)
6. }
7. }
8. }

## Scala for- loop Example for Iterating Collection

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** list = List(1,2,3,4,5,6,7,8,9)          // Creating a list
4. **for**( i <- list){                         // Iterating the list
5. println(i)
6. }
8. }
9. }

## Scala for-each loop Example for Iterating Collection

In the below code we have use three approaches of for-each loop. You can implement any of them according to your need.

1. **object** MainObject {
2. **def** main(args: Array[String]) {
3. **var** list = List(1,2,3,4,5,6,7,8,9)  // Creating a list
4. list.foreach{
5. println     // Print each element
6. }
7. list.foreach(print)
8. println
9. list.foreach((element:**Int**)=>print(element+" "))      // Explicitly mentioning type of elements
10. }
11. }

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

In the above example, we have used *by* keyword. The *by* keyword is used to skip the iteration. When you code like: by 2 it means, this loop will skip all even iterations of loop.

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

## Scala Break Example: Break inner Loop

You can use break statement to terminate execution of inner loop in nested loop.

Let's see an example.

1. **import** scala.util.control.Breaks.\_
2. **object** MainObject {
3. **def** main(args: Array[String]) {
4. **for**(i <- 1 **to** 3){
5. breakable {
6. **for**(j <- 1 **to** 3){
7. **if**(i == 2 & j == 2 )
8. break
9. println(i+" "+j)
10. }
11. }
12. }
13. }
14. }

## 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 Functions

In scala, functions are first class values. You can store function value, pass function as an argument and return function as a value from other function. You can create function by using **def** keyword. You must mention return type of parameters while defining function and return type of a function is optional. If you don't specify return type of a function, default return type is Unit.

## Scala Function Declaration Syntax

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

## 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. }

## 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. }

# Scala Functions

Scala supports functional programming approach. It provides rich set of built-in functions and allows you to create user defined functions also.

In scala, functions are first class values. You can store function value, pass function as an argument and return function as a value from other function. You can create function by using **def** keyword. You must mention return type of parameters while defining function and return type of a function is optional. If you don't specify return type of a function, default return type is Unit.

## 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. }

## 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. }

## 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. }

## 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. }

# Higher Order Functions

## 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. functionExample(25, multiplyBy3)
5. }
6. **def** functionExample(a:**Int**, f:**Int**=>AnyVal):Unit = {
7. println(f(a))                                   // Calling that function
8. }
9. **def** multiplyBy2(a:**Int**):**Int** = {
10. a\*2
11. }
12. **def** multiplyBy3a:**Int**):**Int** = {
13. a\*3
14. }
15. }

## 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. }

## 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. }

## 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. }

## Scala Example Multiline Expression

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

## 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. }

## 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. }

## 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);

# Nested Methods

In Scala it is possible to nest function definitions. The following object provides a factorialfunction for computing the factorial of a given number:

1. def factorial(x: Int): Int = {
2. def fact(x: Int, accumulator: Int): Int = {
3. if (x <= 1) accumulator
4. else fact(x - 1, x \* accumulator)
5. }
6. fact(x, 1)
7. }
8. println("Factorial of 2: " + factorial(2))
9. println("Factorial of 3: " + factorial(3))

# Higher-order Functions

Scala allows the definition of higher-order functions. These are functions that take other functions as parameters, or whose result is a function. Here is a function apply which takes another function fand a value v and applies function f to v:

1. def apply(f: Int => String, v: Int) = f(v)

Note: methods are automatically coerced to functions if the context requires this.

Here is another example:

1. class Decorator(left: String, right: String) {
2. def layout[A](x: A) = left + x.toString() + right
3. }
4. object FunTest extends App {
5. def apply(f: Int => String, v: Int) = f(v)
6. val decorator = new Decorator("[", "]")
7. println(apply(decorator.layout, 7))
8. }

Execution yields the output:

1. [7]

In this example, the method decorator.layout is coerced automatically to a value of type Int => String as required by method apply. Please note that method decorator.layout is a polymorphic method (i.e. it abstracts over some of its signature types) and the Scala compiler has to instantiate its method type first appropriately.

# Classes

Classes in Scala are blueprints for creating objects. They can contain methods, values, variables, types, objects, traits, and classes which are collectively called members. Types, objects, and traits will be covered later in the tour.

## Defining a class

A minimal class definition is simply the keyword class and an identifier. Class names should be capitalized.

1. class User
2. val user1 = new User

The keyword new is used to create an instance of the class. User has a default constructor which takes no arguments because no constructor was defined. However, you’ll often want a constructor and class body. Here is an example class definition for a point:

1. class Point(var x: Int, var y: Int) {
2. def move(dx: Int, dy: Int): Unit = {
3. x = x + dx
4. y = y + dy
5. }
6. override def toString: String =
7. s"($x, $y)"
8. }
9. val point1 = new Point(2, 3)
10. point1.x // 2
11. println(point1) // prints (x, y)

This Point class has four members: the variables x and y and the methods move andtoString. Unlike many other languages, the primary constructor is in the class signature (var x: Int, var y: Int). The move method takes two integer arguments and returns the Unit value (), which carries no information. This corresponds roughly with void in Java-like languages. toString, on the other hand, does not take any arguments but returns a String value. Since toString overrides toString from AnyRef, it is tagged with the override keyword.

## Constructors

Constructors can have optional parameters by providing a default value like so:

1. class Point(var x: Int = 0, var y: Int = 0)
2. val origin = new Point // x and y are both set to 0
3. val point1 = new Point(1)
4. println(point1.x) // prints 1

In this version of the Point class, x and y have the default value 0 so no arguments are required. However, because the constructor reads arguments left to right, if you just wanted to pass in a y value, you would need to name the parameter.

1. class Point(var x: Int = 0, var y: Int = 0)
2. val point2 = new Point(y=2)
3. println(point2.y) // prints 2

This is also a good practice to enhance clarity.

## Private Members and Getter/Setter Syntax

Members are public by default. Use the private access modifier to hide them from outside of the class.

1. class Point {
2. private var \_x = 0
3. private var \_y = 0
4. private val bound = 100
5. def x = \_x
6. def x\_= (newValue: Int): Unit = {
7. if (newValue < bound) \_x = newValue else printWarning
8. }
9. def y = \_y
10. def y\_= (newValue: Int): Unit = {
11. if (newValue < bound) \_y = newValue else printWarning
12. }
13. private def printWarning = println("WARNING: Out of bounds")
14. }
15. val point1 = new Point
16. point1.x = 99
17. point1.y = 101 // prints the warning

In this version of the Point class, the data is stored in private variables \_x and \_y. There are methods def x and def y for accessing the private data. def x\_= and def y\_= are for validating and setting the value of \_x and \_y. Notice the special syntax for the setters: the method has \_= appended to the identifier of the getter and the parameters come after.

Primary constructor parameters with val and var are public. However, because vals are immutable, you can’t write the following.

1. class Point(val x: Int, val y: Int)
2. val point = new Point(1, 2)
3. point.x = 3 // <-- does not compile

Parameters without val or var are private values, visible only within the class.

1. class Point(x: Int, y: Int)
2. val point = new Point(1, 2)
3. point.x // <-- does not compile

# Traits

Traits are used to share interfaces and fields between classes. They are similar to Java 8’s interfaces. Classes and objects can extend traits but traits cannot be instantiated and therefore have no parameters.

## Defining a trait

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

1. trait HairColor

Traits become especially useful as generic types and with abstract methods.

1. trait Iterator[A] {
2. def hasNext: Boolean
3. def next(): A
4. }

Extending the trait Iterator[A] requires a type A and implementations of the methods hasNext and next.

## Using traits

Use the extends keyword to extend a trait. Then implement any abstract members of the trait using the override keyword:

1. trait Iterator[A] {
2. def hasNext: Boolean
3. def next(): A
4. }
5. class IntIterator(to: Int) extends Iterator[Int] {
6. private var current = 0
7. override def hasNext: Boolean = current < to
8. override def next(): Int = {
9. if (hasNext) {
10. val t = current
11. current += 1
12. t
13. } else 0
14. }
15. }
16. val iterator = new IntIterator(10)
17. iterator.next() // prints 0
18. iterator.next() // prints 1

This IntIterator class takes a parameter to as an upper bound. It extends Iterator[Int]which means that the next method must return an Int.

## Subtyping

Where a given trait is required, a subtype of the trait can be used instead.

1. import scala.collection.mutable.ArrayBuffer
2. trait Pet {
3. val name: String
4. }
5. class Cat(val name: String) extends Pet
6. class Dog(val name: String) extends Pet
7. val dog = new Dog("Harry")
8. val cat = new Cat("Sally")
9. val animals = ArrayBuffer.empty[Pet]
10. animals.append(dog)
11. animals.append(cat)
12. animals.foreach(pet => println(pet.name)) // Prints Harry Sally

The trait Pet has an abstract field name which gets implemented by Cat and Dog in their constructors. On the last line, we call pet.name which must be implemented in any subtype of the trait Pet.

# Class Composition with Mixins

Mixins are traits which are used to compose a class.

1. abstract class A {
2. val message: String
3. }
4. class B extends A {
5. val message = "I'm an instance of class B"
6. }
7. trait C extends A {
8. def loudMessage = message.toUpperCase()
9. }
10. class D extends B with C
11. val d = new D
12. d.message // I'm an instance of class B
13. d.loudMessage // I'M AN INSTANCE OF CLASS B

Class D has a superclass B and a mixin C. Classes can only have one superclass but many mixins (using the keywords extends and with respectively). The mixins and the superclass may have the same supertype.

Now let’s look at a more interesting example starting with an abstract class:

1. abstract class AbsIterator {
2. type T
3. def hasNext: Boolean
4. def next: T
5. }

The class has an abstract type T and the standard iterator methods.

Next, we’ll implement a concrete class (all abstract members T, hasNext, and next have implementations):

1. class StringIterator(s: String) extends AbsIterator {
2. type T = Char
3. private var i = 0
4. def hasNext = i < s.length
5. def next = {
6. val ch = s charAt i
7. i += 1
8. ch
9. }
10. }

StringIterator takes a String and can be used to iterate over the String (e.g. to see if a String contains a certain character).

Now let’s create a trait which also extends AbsIterator.

1. trait RichIterator extends AbsIterator {
2. def foreach(f: T => Unit): Unit = while (hasNext) f(next)
3. }

Because RichIterator is a trait, it doesn’t need to implement the abstract members of AbsIterator.

We would like to combine the functionality of StringIterator and RichIterator into a single class.

1. object StringIteratorTest extends App {
2. class RichStringIter extends StringIterator(args(0)) with RichIterator
3. val iter = new RichStringIter
4. iter foreach println
5. }

The new class RichStringIter has StringIterator as a superclass and RichIterator as a mixin.

With single inheritance we would not be able to achieve this level of flexibility

# Scala String

Scala String Comparison Example

In scala, you can compare two string objects by using == (equal) method. The following program describes how to use equal operator. It returns boolean value either true or 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 == s2)
7. println(s2 == s3)
8. }
9. }
11. **object** MainObject{
12. **def** main(args:Array[String]){
13. **var** s = **new** StringExample()
14. s.show()
15. }
16. }

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. }

## Scala compareTo() Method

## Scala Concatenation Example by using + (Plus) Operator

Some methods are –

* **length** – Returns the number of Unicode code units in a string

**e.g.**

println(I.length)

* **charAt –** It has an index number as an argument and returns the Char at the specified index.

**e.g.**

println(I.charAt(2))

**Output**

t

* **startsWith** – Tests whether this string starts with the argument of this method.

**e.g.**

println(A.startsWith("I"))

**Output**

True

* **endsWith** – Tests whether this string ends with the argument of this method.

**e.g.**

println(I.endsWith("i"))

**Output**

False

* **indexOf –** Returns the index within this string object of the first occurrence of the string argument.

**e.g.**

println(I.indexOf("n"))

**Output**

1

* **substring** – Returns a new string that is a substring of this string. It may take one or two arguments. If it takes one argument then substring begins with the specified index to the end of string and if it takes two argument then first specify the starting of string and second string specify the ending of string

**e.g.**

println(I.substring(1,4))

**Output**

nte

* **contains** – Examine whether a string contains a particular substring or not.

**e.g.**

if(I.contains("inte"))

println("contains specified string")

* **replace** – It takes two arguments from which first argument is replaced by second argument form the specified string.

**e.g.**

println(I.replace('a','t'))

println(I)

* **toLowerCase** – converts all of the characters in this string to lower case.

**e.g.**

println(I.toLowerCase())

* **toUpperCase** – converts all of the characters in this string to upper case.

**e.g.**

println(I.toUpperCase())

## Scala String concat() Method Example

## Scala substring() Method Example

# 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 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. }

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.

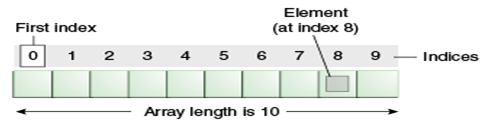
1. **class** ArrayExample{
2. **def** show(arr:Array[**Int**]){
3. **for**(a<-arr)                // Traversing array elements
4. println(a)
5. println("Third Element = "+ arr(2))        // Accessing elements by using index
6. }
7. }
9. **object** MainObject{
10. **def** main(args:Array[String]){
11. **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. }

# 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. }

Scala Multidimensional Array Example by using ofDim

In This example, we have created array by using ofDim method.

1. **class** ArrayExample{
2. **var** arr = Array.ofDim[**Int**](2,2)          // Creating multidimensional array
3. arr(1)(0) = 15                          // Assigning value
4. **def** show(){
5. **for**(i<- 0 **to** 1){                       // Traversing elements by using loop
6. **for**(j<- 0 **to** 1){
7. print(" "+arr(i)(j))
8. }
9. println()
10. }
11. println("Third Element = "+ arr(1)(1))        // Accessing elements by using index
12. }
13. }
15. **object** MainObject{
16. **def** main(args:Array[String]){
17. **var** a = **new** ArrayExample()
18. a.show()
19. }
20. }

File Handing

## ****Overview of reading and writing file****oprations

File operations mostly includes reading data from file or writing data into files.

**Writing Data form File**

**e.g.**

import java.io.\_

object Test {

def main(args: Array[String]) {

val writer = new PrintWriter(new File("test.txt" ))

writer.write("Hello test")

writer.close()

}

}

**Reading Data from File**

**e.g.**

import scala.io.Source

object Test {

def main(args: Array[String]) {

println("Data in File is:" )

Source.fromFile("test.txt" ).foreach{

print

}

}

}

## ****Scala Collections****

## ****Introduction to Scala Collections****

Collections are the container of things which contains random number of elements. All collection classes are found in the package scala.collection. Collections are of two types –

* Mutable Collections
* Immutable Collections

**Mutable Collection –**This type of collection is changed after it is created. All Mutable collection classes are found in the package scala.collection.mutable.

**Immutable Collection** – This type of collection will never change after it is created. All Immutable collection classes are found in the package scala.collection.immutable.

Most Common Collection types are –

* List
* Map
* Set
* Tuple
* Iterators

**1. List –**List is a collection of similar types of elements which are immutable. It represents the Linked list. If list contains t tpes of elements then it can be represented as –

List[t]

The empty list is specified by Nil which is an object that represents any empty list. The method :: pronounced cons transforms an object and a list into a new list whose head is the object and whose tail is the first list.

val numbers: List[Int] = List(10, 20, 30 ,40)  //List of Integers

val empty: List[Int] = List()  // Empty List

val twodim: List[List[Int]] =

List (

List (0, 1, 0),

List (1, 1, 0)

)

**List Constructors –**All lists are built from two more fundamental constructors that is Nil and :: .Nil represents an empty list. The infix operator :: expresses list extension. That is, x :: xs represents a list whose first element is x, which is followed by list xs. You can also define the list as follows:

val numbers = 10 :: (20 :: (30 :: (40 :: Nil)))

val empty = Nil

val twodim = (0 :: (1 :: (0 :: Nil))) ::

(1 :: (1 :: (0 :: Nil))) :: Nil

For simplicity you can definje above list as follows:-

val numbers = 10 :: 20 :: 30 :: 40 :: Nil

**Basic operations on lists –**

|  |  |
| --- | --- |
| **Methods** | **Description** |
| head | It returns the first element of a list. |
| tail | It returns a list consisting of all elements except the first. |
| isEmpty | It returns true if the list is empty. |

**e.g.**

object Test {

def main(args: Array[String]) {

**val** numbers = 10 :: (20 :: (30 :: (40 :: Nil)))

val data = Nil

println( "Head of Number is : " + numbers.head )

println( "Tail of Number is : " + numebrs.tail )

println( "Check number is empty or not : " + numbers.isEmpty )

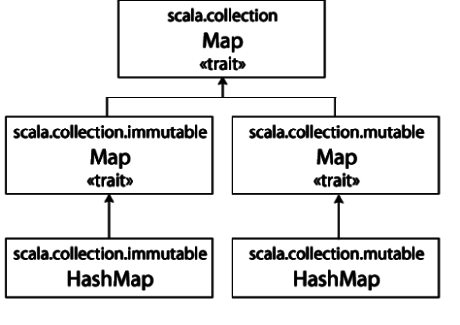
println( "Check data is empty or not : " + data.isEmpty )

}

}

* **List.concat() –**  It is used to perform concatenation of two lists.
* **List.fill() –** It creates a lsit which contains same element.
* **List.tabulate() –** It converts the list in tabular form.
* **List.reverse –** It is used to reverse the list elements.

**2. Map (Hash Table)**– It is a collection of key/value pairs where key are unique and value is retrieved according to the key. Scala Online Training provides mutable and immutable versions of Map. If you want to use immutable map then use Map and if you want to use mutable map the use mutable.Map.

[](https://cdn.intellipaat.com/wp-content/uploads/2015/11/class-hierarchy-for-scala-map.png)

**e.g.**

var I:Map[String ,Int] = Map()  // Creates empty hash table whose values are integers and keys are string type

val data = Map(

‘a’ ->10,

‘b’ -> 20,

‘c’->30,

‘d’ ->40

)

**Basic Operations on Map:**

All operations on maps can be expressed in terms of the following three methods:

|  |  |
| --- | --- |
| **Methods** | **Description** |
| keys | It returns an iterable containing each key in the map. |
| values | It returns an iterable containing each value in the map. |
| isEmpty | It returns true if the map is empty otherwise false. |

**e.g.**

object Test {

def main(args: Array[String]) {

val numbers = Map(

‘a’ ->10, ‘b’ -> 20, ‘c’->30, ‘d’ ->40

)

val data: Map[char, Int] = Map()

println( "Keys in Numbers are : " + numbers.keys )

println( "Values in numbers are : " + numbers.values )

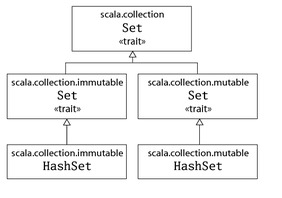
println( "Check number is empty or not : " + numbers.isEmpty )

println( "Check data is empty or not : " + data.isEmpty )

}

}

**3. Set** – It is a collection of elements which are of same type but does not contain same elements. By default scala uses immutable set. Scala provides mutable and immutable versions of Set. If you want to use immutable set then use Set and if you want to use mutable set the use mutable.Set.

[](https://cdn.intellipaat.com/wp-content/uploads/2015/11/class-hierarchy-for-scala-set.png)

var i : Set[Int] = Set()    // Empty set of integer type

var i : Set[Int] = Set(10, 20, 30, 40)  // Set of integer type

**Basic Operations on Set:**

All operations on sets can be expressed in terms of the following three methods:

|  |  |
| --- | --- |
| **Methods** | **Description** |
| head | It returns the first element of a set. |
| tail | It returns a set consisting of all elements except the first. |
| isEmpty | It returns true if the set is empty otherwise false. |

**e.g.**

object Test {

def main(args: Array[String]) {

val numbers = Set(10, 20, 30, 40)

val data: Set[Int] = Set()

println( "Head of Number is : " + numbers.head )

println( "Tail of numbers is : " + numbers.tail )

println( "Check number is empty or not : " + numbers.isEmpty )

println( "Check data is empty or not : " + data.isEmpty )

}

}

Check data is empty or not : True

* **Set.++() –**It is used to concatenate the two or more sets.
* **Set.min –**It is used to find the minimum value from the elements of the set.
* **Set.max() –**It is used to find the maximum value from the elements of the set.
* **Set.intersect –**It is used to find the common values between two sets.
* **Set.& –**It is also used to find the common values between two sets.

**4. Tuples** – It is a collection of heterogeneous types of objects that is different types of objects which combines a fixed number of items together.

Tuple that contains an Int and a String:

val i = (1, "Test")

To access tuple elements ‘.\_’ is used.

**e.g.**

object Test {

def main(args: Array[String]) {

val i = (10, 20, 30, 40)

val total = i.\_1 + i.\_2 + i.\_3 + i.\_4

println( "Total of Elements is : "  + total)

}

}

* **Tuple.productIterator() –**It is used to iterate over all the elements of a Tuple.
* **Tuple.toString() –**It is used to concatenate all the elements of the tuple into a string.
* **Tuple.swap –**It is used to swap the elements of a Tuple.

**5. Iterator –** It is used to access the elements of the collection one by one.Two important operations of iterator is –

* next
* hasNext

**next –** It is used to return the next element of the iterator and move ahead the state of the iterator.

**hasNext –**It is used to find out whether there are more elements to return.

**e.g.**

object Test {

def main(args: Array[String]) {

val i = Iterator("hello", "Test", "a", "ecommerce", “site”)

while (i.hasNext){

println(i.next())

}

}

}