What is Scala

Scala is a general-purpose programming language. It supports object oriented, functional and imperative programming approaches. It is a strong static type language. In scala, everything is an object whether it is a function or a number. It does not have concept of primitive data.

It was designed by Martin Odersky. It was officially released for java platform in early 2004 and for .Net framework in June 2004. Later on, Scala dropped .Net support in 2012.

Scala is influenced by Java, Haskell, Lisp, Pizza etc. and influenced to F#, Fantom, Red etc.

File extension of scala source file may be either .scala or .sc.

You can create any kind of application like web application, enterprise application, mobile application, desktop based application etc.

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. }

Where to use Scala

* Web applications
* Utilities and libraries
* Data streaming with Akka
* Parallel batch processing
* Concurrency and distributed application
* Data analysis with Spark
* AWS lambda expression
* Ad hoc scripting in REPL etc.

In Scala, you can create any type of application in less time and coding whether it is web based, mobile based or desktop based application. Scala provides you powerful tools and API by using which you can create applications. Here, You can use play framework which provides a platform to build web application rapidly.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**next →**](http://www.javatpoint.com/features-of-scala)[**← prev**](http://www.javatpoint.com/what-is-scala) History of Scala Scala History of scala 1  Scala is a general purpose programming language. It was created and developed by Martin Odersky. Martin started working on Scala in 2001 at the Ecole Polytechnique Federale de Lausanne (EPFL). It was officially released on January 20, 2004.  Scala is not an extension of Java, but it is completely interoperable with it. While compilation, Scala file translates to Java bytecode and runs on JVM (Java Virtual machine).  Scala was designed to be both object-oriented and functional. It is a pure object-oriented language in the sense that every value is an object and functional language in the sense that every function is a value. The name of scala is derived from word scalable which means it can grow with the demand of users. Versions of Scala  |  |  | | --- | --- | | **Version** | **Released date** | | 2.0 | 12-Mar-2006 | | 2.1.8 | 23-Aug-2006 | | 2.3.0 | 23-Nov-2006 | | 2.4.0 | 09-Mar-2007 | | 2.5.0 | 02-May-2007 | | 2.6.0 | 27-Jul-2007 | | 2.7.0 | 07-Feb-2008 | | 2.8.0 | 14-Jul-2010 | | 2.9.0 | 12-May-2011 | | 2.10 | 04-Jan-2013 | | 2.10.2 | 06-Jun-2013 | | 2.10.3 | 01-Oct-2013 | | 2.10.4 | 18-Mar-2014 | | 2.10.5 | 05-Mar-2015 | | 2.11.0 | 21-Apr-2014 | | 2.11.1 | 20-May-2014 | | 2.11.2 | 22-Jul-2014 | | 2.11.4 | 31-Oct-2014 | | 2.11.5 | 08-Jan-2015 | | 2.11.6 | 05-Mar-2015 | | 2.11.7 | 23-Jun-2015 | | 2.11.8 | 8-Mar-2016 | | 2.12.1 | 5-December - 2016 |  **Popularity of Scala**  * Twitter have announced that it had switched large portions of its backend from Ruby to Scala and intended to convert the rest. * Apple Inc. uses Scala in certain teams, along with Java and the Play framework. * The New York Times revealed in 2014 that its internal content management system Blackbeard is built using Scala, Akka and Play Framework. * There are teams within Google that use Scala, mostly due to acquisitions such as Firebase and Nest. * The Walmart Canada Uses Scala for their back end platform. |

# Features of Scala

There are following features of scala:

* Type inference
* Singleton object
* Immutability
* Lazy computation
* Case classes and Pattern matching
* Concurrency control
* String interpolation
* Higher order function
* Traits
* Rich collection set

### **Type Inference**

In Scala, you don't require to mention data type and function return type explicitly. Scala is enough smart to deduce the type of data. The return type of function is determined by the type of last expression present in the function.

### **Singleton object**

In Scala, there are no static variables or methods. Scala uses singleton object, which is essentially class with only one object in the source file. Singleton object is declared by using object instead of class keyword.

### **Immutability**

Scala uses immutability concept. Each declared variable is immutable by default. Immutable means you can't modify its value. You can also create mutable variables which can be changed.

Immutable data helps to manage concurrency control which requires managing data.

### **Lazy Computation**

In Scala, computation is lazy by default. Scala evaluates expressions only when they are required. You can declare a lazy variable by using lazy keyword. It is used to increase performance.

### **Case classes and Pattern matching**

Scala case classes are just regular classes which are immutable by default and decomposable through pattern matching.

All the parameters listed in the case class are public and immutable by default.

Case classes support pattern matching. So, you can write more logical code.

### **Concurrency control**

Scala provides standard library which includes the actor model. You can write concurrency code by using actor. Scala provides one more platform and tool to deal with concurrency known as Akka. Akka is a separate open source framework that provides actor-based concurrency. Akka actors may be distributed or combined with software transactional memory.

### **String Interpolation**

Since Scala 2.10.0, Scala offers a new mechanism to create strings from your data. It is called string interpolation. String interpolation allows users to embed variable references directly in processed string literals. Scala provides three string interpolation methods: s, f and raw.

### **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 another function is called higher order function.

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

### **Traits**

A trait is like an interface with a partial implementation. In Scala, trait is a collection of abstract and non-abstract methods. You can create trait that can have all abstract methods or some abstract and some non-abstract methods.

Traits are compiled into Java interfaces with corresponding implementation classes that hold any methods implemented in the traits.

### **Rich Set of Collection**

Scala provides rich set of collection library. It contains classes and traits to collect data. These collections can be mutable or immutable. You can use it according to your requirement. Scala.collection.mutable package contains all the mutable collections. You can add, remove and update data while using this package.

Scala.collection.immutable package contains all the immutable collections. It does not allow you to modify data.

# Simple Program of Scala

In this tutorial, you will learn how to write scala programs. To write scala program you need to install scala on your machine. You must have latest jdk installed on your machine because scala compiler creates .class file which is a byte code. Scala interpreter executes this byte code by using jvm (Java Virtual Machine).

## Scala Example: Hello Scala

The following code example is a simple scala program.

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

In the above code, we have created an object ScalaExample. It contains a main method and display message using println method.

This file is saved with the name **ScalaExample.scala.**

Command to compile this code is: **scalac ScalaExample.scala**

Command to execute the compiled code is: **scala ScalaExample**

After executing code it yields the following output.

Output:

Hello Scala

You can also use IDE (Integrated Development Environment) for executing scala code.

The above example is written using object oriented approach. You can also use functional approach to write code in scala.

### **Scala Example: Hello Scala**

Below is the example by using functional approach.

1. **def** scalaExample{
2. println("Hello Scala")
3. }
4. scalaExample            // Calling of function

Output:

Hello Scala

# 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 |

**Next Topic**[Scala Conditional Expressions](http://www.javatpoint.com/scala-conditional-expressions)

# 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

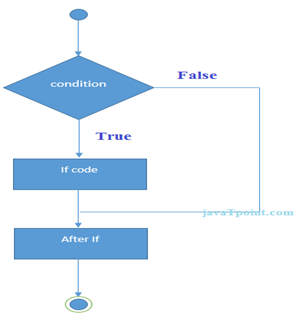
## Scala if statement

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. }

### **Flowchart**



## 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

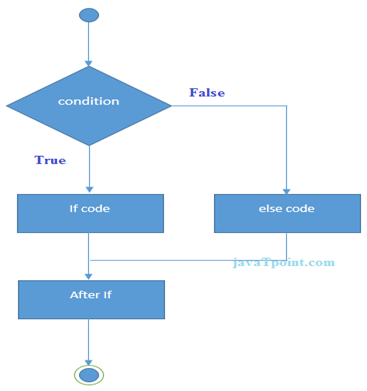
## Scala If-Else Statement

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. }

### **Flowchart**



## 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

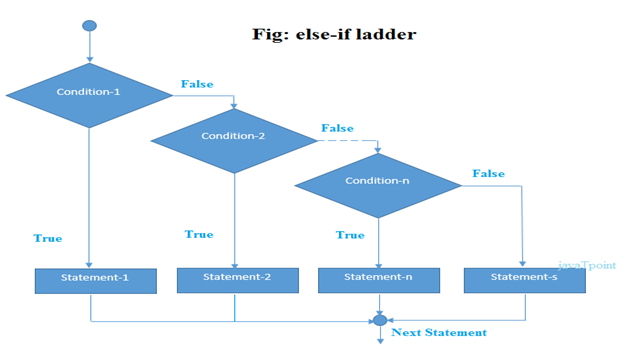
## Scala If-Else-If Ladder Statement

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. }

### **Flowchart**



## 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. }

Output:

-1

Scala Pattern Matching

Pattern matching is a feature of scala. It works same as switch case in other programming languages. It matches best case available in the pattern.

Let's see an example.

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. }

Output:

Hello

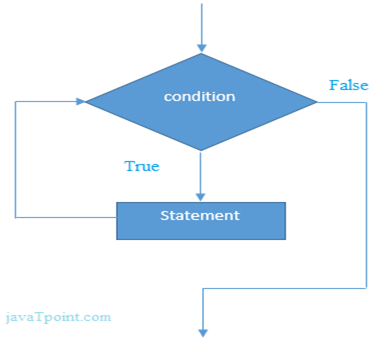
# Scala while loop

In Scala, while loop is used to iterate code till the specified condition. It tests boolean expression and iterates again and again. You are recommended to use while loop if you don't know number of iterations prior.

**Syntax**

1. **while**(boolean expression){
2. // Statements to be executed
3. }

### **Flowchart:**



## 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. }

Output:

10

12

14

16

?

Ctrl+Z // To stop execution of program

Scala for loop

In scala, *for loop* is known as for-comprehensions. It can be used to iterate, filter and return an iterated collection. The for-comprehension looks a bit like a for-loop in imperative languages, except that it constructs a list of the results of all iterations.

**Syntax**

1. **for**( i <- range){
2. // statements to be executed
3. }

In the above syntax, range is a value which has *start* and *end* point. You can pass range by using **to** or **until** keyword.

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. }

Output:

1

2

3

4

5

6

7

8

9

10

In the below example, *until* is used instead of *to*. The major difference between *until* and *to* is, *to* includes start and end value given in the range, while *until* excludes last value of the range. So, the below example will print only 1 to 9.

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. }

Output:

1

2

3

4

5

6

7

8

9

It is helpful to apply *until* keyword when you are iterating string or array, because array range is 0 to n-1. *until* does not exceed to n-1. So, your code will not complain of upper range.

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. }

Output:

2

4

6

8

10

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. }

Output:

1

2

3

4

5

6

7

8

9

10

Scala for-loop in Collection

In scala, you can iterate collections like list, sequence etc, either by using for each loop or for-comprehensions.

Let's see an example.

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. }

Output:

1

2

3

4

5

6

7

8

9

10

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. }

Output:

1

2

3

4

5

6

7

8

9

123456789

1 2 3 4 5 6 7 8 9

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. }

Output:

1

3

5

7

9

Scala Break

Break is used to break a loop or program execution. It skips the current execution. Inside inner loop it breaks the execution of inner loop.

In scala, there is no break statement but you can do it by using break method and by importing *scala.util.control.Breaks.\_* package. Let's see an example.

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. }

Output:

1

3

5

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. }

Output:

1 1

1 2

1 3

2 1

3 1

3 2

3 3

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. }

Output:

1

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. }

Output:

1

Scala Documentation Comment Example

1. // Example of documentation 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. \*/
12. }

Output:

1

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. }

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. }

Output:

17

17

17

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);

Output:

45

**Next Topic**[Scala Object and Class](http://www.javatpoint.com/scala-object-and-class)

Scala Object and Class

Unlike java, scala is a pure object oriented programming language. It allows us to create object and class so that you can develop object oriented applications.

Object

Object is a real world entity. It contains state and behavior. Laptop, car, cell phone are the real world objects. Object typically has two characteristics:

**1) State**: data values of an object are known as its state.

**2) Behavior**: functionality that an object performs is known as its behavior.

Object in scala is an instance of class. It is also known as runtime entity.

Class

Class is a template or a blueprint. It is also known as collection of objects of similar type.

In scala, a class can contain:

1. Data member
2. Member method
3. Constructor
4. Block
5. Nested class
6. Super class information etc.

You must initialize all instance variables in the class. There is no default scope. If you don't specify access scope, it is public. There must be an object in which main method is defined. It provides starting point for your program. Here, we have created an example of class.

Scala Sample Example of Class

1. **class** Student{
2. **var** id:**Int** = 0;                         // All fields must be initialized
3. **var** name:String = **null**;
4. }
5. **object** MainObject{
6. **def** main(args:Array[String]){
7. **var** s = **new** Student()               // Creating an object
8. println(s.id+" "+s.name);
9. }
10. }

Output:

0 null

Scala Sample Example2 of Class

In scala, you can create class like this also. Here, constructor is created in class definition. This is called primary constructor.

1. **class** Student(id:**Int**, name:String){     // Primary constructor
2. **def** show(){
3. println(id+" "+name)
4. }
5. }
6. **object** MainObject{
7. **def** main(args:Array[String]){
8. **var** s = **new** Student(100,"Martin")   // Passing values to constructor
9. s.show()                // Calling a function by using an object
10. }
11. }

Output:

100 Martin

Scala Example of class that maintains the records of students

1. **class** Student(id:**Int**, name:String){
2. **def** getRecord(){
3. println(id+" "+name);
4. }
5. }
7. **object** MainObject{
8. **def** main(args: Array[String]){
9. **var** student1 = **new** Student(101,"Raju");
10. **var** student2 = **new** Student(102,"Martin");
11. student1.getRecord();
12. student2.getRecord();
13. }
14. }

Output:

101 Raju

102 Martin

Scala Anonymous object

In scala, you can create anonymous object. An object which has no reference name is called anonymous object. It is good to create anonymous object when you don't want to reuse it further.

Scala Anonymous object Example

1. **class** Arithmetic{
2. **def** add(a:**Int**, b:**Int**){
3. **var** add = a+b;
4. println("sum = "+add);
5. }
6. }
8. **object** MainObject{
9. **def** main(args:Array[String]){
10. **new** Arithmetic().add(10,10);
12. }
13. }

Output:

Sum = 20

# Scala Singleton and Companion Object

## Scala Singleton Object

Singleton object is an object which is declared by using object keyword instead by class. No object is required to call methods declared inside singleton object.

In scala, there is no static concept. So scala creates a singleton object to provide entry point for your program execution.  
If you don't create singleton object, your code will compile successfully but will not produce any output. Methods declared inside Singleton Object are accessible globally. A singleton object can extend classes and traits.

### **Scala Singleton Object Example**

1. **object** Singleton{
2. **def** main(args:Array[String]){
3. SingletonObject.hello()         // No need to create object.
4. }
5. }

8. **object** SingletonObject{
9. **def** hello(){
10. println("Hello, This is Singleton Object")
11. }
12. }

Output:

Hello, This is Singleton Object

## Scala Companion Object

In scala, when you have a class with same name as singleton object, it is called companion class and the singleton object is called companion object.

The companion class and its companion object both must be defined in the same source file.

### **Scala Companion Object Example**

1. **class** ComapanionClass{
2. **def** hello(){
3. println("Hello, this is Companion Class.")
4. }
5. }
6. **object** CompanoinObject{
7. **def** main(args:Array[String]){
8. **new** ComapanionClass().hello()
9. println("And this is Companion Object.")
10. }
11. }

Output:

Hello, this is Companion Class.

And this is Companion Object.

# Scala Case Classes and Case Object

Scala case classes are just regular classes which are immutable by default and decomposable through pattern matching.

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.

**Syntax**

1. **case** **class** className(parameters)

### **Scala Case Class Example**

1. **case** **class** CaseClass(a:**Int**, b:**Int**)
3. **object** MainObject{
4. **def** main(args:Array[String]){
5. **var** c =  CaseClass(10,10)       // Creating object of case class
6. println("a = "+c.a)               // Accessing elements of case class
7. println("b = "+c.b)
8. }
9. }

Output:

a = 10

b = 10

Case classes support pattern matching. So, you can use that in patterns. Following is the example of case classes and pattern.

## Scala Case Class and Pattern Matching Example

A case class which has no arguments is declared as case object instead of case class. case object is serializeable by default.

1. **trait** SuperTrait
2. **case** **class** CaseClass1(a:**Int**,b:**Int**) **extends** SuperTrait
3. **case** **class** CaseClass2(a:**Int**) **extends** SuperTrait         // Case class
4. **case** **object** CaseObject **extends** SuperTrait               // Case object
5. **object** MainObject{
6. **def** main(args:Array[String]){
7. callCase(CaseClass1(10,10))
8. callCase(CaseClass2(10))
9. callCase(CaseObject)
10. }
11. **def** callCase(f:SuperTrait) = f **match**{
12. **case** CaseClass1(f,g)=>println("a = "+f+" b ="+g)
13. **case** CaseClass2(f)=>println("a = "+f)
14. **case** CaseObject=>println("No Argument")
15. }
16. }

Output:

a = 10 b =10

a = 10

No Argument

Scala Constructor

In scala, constructor is not special method. Scala provides primary and any number of auxiliary constructors. We have explained each in details in the following example.

Scala Default Primary Constructor

In scala, if you don't specify primary constructor, compiler creates a constructor which is known as primary constructor. All the statements of class body treated as part of constructor. It is also known as default constructor.

Scala Default Primary Constructor Example

1. **class** Student{
2. println("Hello from default constructor");
3. }

Output:

Hello from default constructor

Scala Primary Constructor

Scala provides a concept of primary constructor with the definition of class. You don't need to define explicitly constructor if your code has only one constructor. It helps to optimize code. You can create primary constructor with zero or more parameters.

Scala Primary Constructor Example

1. **class** Student(id:**Int**, name:String){
2. **def** showDetails(){
3. println(id+" "+name);
4. }
5. }
7. **object** MainObject{
8. **def** main(args:Array[String]){
9. **var** s = **new** Student(101,"Rama");
10. s.showDetails()
11. }
12. }

Output:

101 Rama

Scala Secondary (auxiliary) Constructor

You can create any number of auxiliary constructors in a class. You must call primary constructor from inside the auxiliary constructor. this keyword is used to call constructor from other constructor. When calling other constructor make it first line in your constructor.

Scala Secondary Constructor Example

1. **class** Student(id:**Int**, name:String){
2. **var** age:**Int** = 0
3. **def** showDetails(){
4. println(id+" "+name+" "+age)
5. }
6. **def** this(id:**Int**, name:String,age:**Int**){
7. this(id,name)       // Calling primary constructor, and it is first line
8. this.age = age
9. }
10. }
12. **object** MainObject{
13. **def** main(args:Array[String]){
14. **var** s = **new** Student(101,"Rama",20);
15. s.showDetails()
16. }
17. }

Output:

101 Rama 20

Scala Example: Constructor Overloading

In scala, you can overload constructor. Let's see an example.

1. **class** Student(id:**Int**){
2. **def** this(id:**Int**, name:String)={
3. this(id)
4. println(id+" "+name)
5. }
6. println(id)
7. }
9. **object** MainObject{
10. **def** main(args:Array[String]){
11. **new** Student(101)
12. **new** Student(100,"India")
13. }
14. }

Output:

101

100

100 India

# Scala Method Overloading

Scala provides method overloading feature which allows us to define methods of same name but having different parameters or data types. It helps to optimize code.

### **Scala Method Overloading Example by using Different Parameters**

In the following example, we have define two add methods with different number of parameters but having same data type.

1. **class** Arithmetic{
2. **def** add(a:**Int**, b:**Int**){
3. **var** sum = a+b
4. println(sum)
5. }
6. **def** add(a:**Int**, b:**Int**, c:**Int**){
7. **var** sum = a+b+c
8. println(sum)
9. }
10. }
12. **object** MainObject{
13. **def** main(args:Array[String]){
14. **var** a  = **new** Arithmetic();
15. a.add(10,10);
16. a.add(10,10,10);
17. }
18. }

Output:

20

30

## Scala Method Overloading Example by using Different Data Type

In the following example, we have created two add method having same number of parameters but different data types.

1. **class** Arithmetic{
2. **def** add(a:**Int**, b:**Int**){
3. **var** sum = a+b
4. println(sum)
5. }
6. **def** add(a:**Double**, b:**Double**){
7. **var** sum = a+b
8. println(sum)
9. }
10. }
11. **object** MainObject{
12. **def** main(args:Array[String]){
13. **var** b = **new** Arithmetic()
14. b.add(10,10)
15. b.add(10.0,20.0)
17. }
18. }

Output:

20

30.0

# Scala this

In scala, this is a keyword and used to refer current object. You can call instance variables, methods, constructors by using this keyword.

### **Scala this Example**

In the following example, *this* is used to call instance variables and primary constructotr.

1. **class** ThisExample{
2. **var** id:**Int** = 0
3. **var** name: String = ""
4. **def** this(id:**Int**, name:String){
5. this()
6. this.id = id
7. this.name = name
8. }
9. **def** show(){
10. println(id+" "+name)
11. }
12. }
14. **object** MainObject{
15. **def** main(args:Array[String]){
16. **var** t = **new** ThisExample(101,"Martin")
17. t.show()
18. }
19. }

Output:

101 Martin

## Scala Constructor Calling by using *this* keyword

In the following example this is used to call constructor. It illustrates how we can call constructor from other constructor. You must make sure that this must be first statement in the constructor while calling to other constructor otherwise compiler throws an error.

1. **class** Student(name:String){
2. **def** this(name:String, age:**Int**){
3. this(name)
4. println(name+" "+age)
5. }
6. }
8. **object** MainObject{
9. **def** main(args:Array[String]){
10. **var** s = **new** Student("Rama",100)
11. }
12. }

Output:

Rama 100

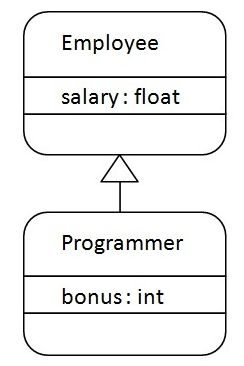
# Scala Inheritance

Inheritance is an object oriented concept which is used to reusability of code. You can achieve inheritance by using extends keyword. To achieve inheritance a class must extend to other class. A class which is extended called super or parent class. a class which extends class is called derived or base class.

**Syntax**

1. **class** SubClassName **extends** SuperClassName(){
2. /\* Write your code
3. \*  methods and fields etc.
4. \*/
5. }

### **Understand the Simple Example of Inheritance**



## Scala Single Inheritance Example

1. **class** Employee{
2. **var** salary:Float = 10000
3. }
5. **class** Programmer **extends** Employee{
6. **var** bonus:**Int** = 5000
7. println("Salary = "+salary)
8. println("Bonus = "+bonus)
9. }
11. **object** MainObject{
12. **def** main(args:Array[String]){
13. **new** Programmer()
14. }
15. }

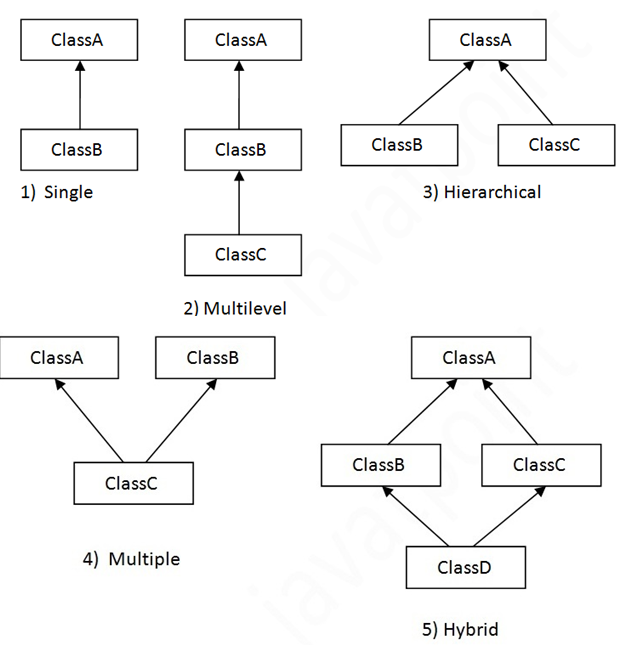
Output:

Salary = 10000.0

Bonus = 5000

## Types of Inheritance in Scala

Scala supports various types of inheritance including single, multilevel, multiple, and hybrid. You can use single, multilevel and hierarchal in your class. Multiple and hybrid can only be achieved by using traits. Here, we are representing all types of inheritance by using pictorial form.



## Scala Multilevel Inheritance Example

1. **class** A{
2. **var** salary1 = 10000
3. }
5. **class** B **extends** A{
6. **var** salary2 = 20000
7. }
9. **class** C **extends** B{
10. **def** show(){
11. println("salary1 = "+salary1)
12. println("salary2 = "+salary2)
13. }
14. }
16. **object** MainObject{
17. **def** main(args:Array[String]){{
18. **var** c = **new** C()
19. c.show()
21. }
22. }

Output:

salary1 = 10000

salary2 = 20000

# Scala Method Overriding

When a subclass has the same name method as defined in the parent class, it is known as method overriding. When subclass wants to provide a specific implementation for the method defined in the parent class, it overrides method from parent class.

In scala, you must use either override keyword or override annotation to override methods from parent class.

## Scala Method Overriding Example 1

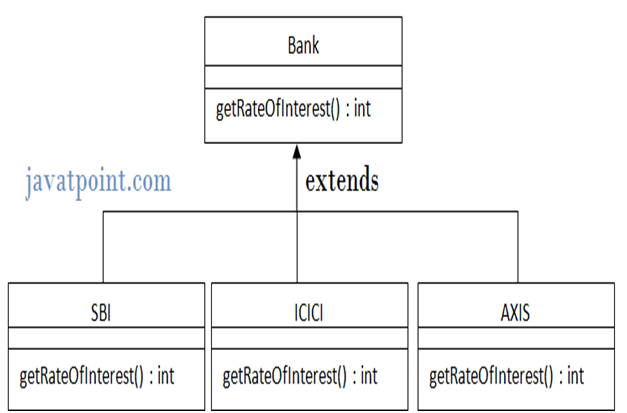
1. **class** Vehicle{
2. **def** run(){
3. println("vehicle is running")
4. }
5. }
7. **class** Bike **extends** Vehicle{
8. override **def** run(){
9. println("Bike is running")
10. }
11. }
13. **object** MainObject{
14. **def** main(args:Array[String]){
15. **var** b = **new** Bike()
16. b.run()
17. }
18. }

Output:

Bike is running

## Real example of method overriding

### **Flowchart**



## Scala Method Overriding Example 2

This example shows how subclasses override the method of parent class.

1. **class** Bank{
2. **def** getRateOfInterest()={
3. 0
4. }
5. }
7. **class** SBI **extends** Bank{
8. override **def** getRateOfInterest()={
9. 8
10. }
11. }
13. **class** ICICI **extends** Bank{
14. override **def** getRateOfInterest()={
15. 7
16. }
17. }
19. **class** AXIS **extends** Bank{
20. override **def** getRateOfInterest()={
21. 9
22. }
23. }
25. **object** MainObject{
26. **def** main(args:Array[String]){
27. **var** s=**new** SBI();
28. **var** i=**new** ICICI();
29. **var** a=**new** AXIS();
30. println("SBI Rate of Interest: "+s.getRateOfInterest());
31. println("ICICI Rate of Interest: "+i.getRateOfInterest());
32. println("AXIS Rate of Interest: "+a.getRateOfInterest());
33. }
34. }

Output:

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

# Scala Field Overriding

In scala, you can override fields also but it has some rules that need to be followed. Below are some examples that illustrate how to override fields.

### **Scala Field Overriding Example1**

1. **class** Vehicle{
2. **var** speed:**Int** = 60
4. }
5. **class** Bike **extends** Vehicle{
6. **var** speed:**Int** = 100
7. **def** show(){
8. println(speed)
9. }
10. }
11. **object** MainObject{
12. **def** main(args:Array[String]){
13. **var** b = **new** Bike()
14. b.show()
15. }
16. }

Output:

Error - variable speed needs 'override' modifier

In scala, you must use either override keyword or override annotation when you are overriding methods or fields of super class. If you don't do this, compiler reports an error and stops execution of program.

## Scala Field Overriding Example2

1. **class** Vehicle{
2. **val** speed:**Int** = 60
4. }
5. **class** Bike **extends** Vehicle{
6. override **val** speed:**Int** = 100     // Override keyword
7. **def** show(){
8. println(speed)
9. }
10. }
11. **object** MainObject{
12. **def** main(args:Array[String]){
13. **var** b = **new** Bike()
14. b.show()
15. }
16. }

Output:

100

In scala, you can override only those variables which are declared by using val keyword in both classes. Below are some interesting examples which demonstrate the whole process.

## Scala Field Overriding Example3

1. **class** Vehicle{
2. **var** speed:**Int** = 60
3. }
4. **class** Bike **extends** Vehicle{
5. override **var** speed:**Int** = 100
6. **def** show(){
7. println(speed)
8. }
9. }
10. **object** MainObject{
11. **def** main(args:Array[String]){
12. **var** b = **new** Bike()
13. b.show()
14. }
15. }

Output:

variable speed cannot override a mutable variable

## Scala Field Overriding Example4

1. **class** Vehicle{
2. **val** speed:**Int** = 60
4. }
6. **class** Bike **extends** Vehicle{
7. override **var** speed:**Int** = 100
8. **def** show(){
9. println(speed)
10. }
11. }
13. **object** MainObject{
14. **def** main(args:Array[String]){
15. **var** b = **new** Bike()
16. b.show()
17. }
18. }

Output:

Error - variable speed needs to be a stable, immutable value

Scala Final

Final is a keyword, which is used to prevent inheritance of super class members into derived class. You can declare final variables, methods and classes also.

Scala Final Variable Example

You can't override final variables in subclass. Let's see an example.

1. **class** Vehicle{
2. **final** **val** speed:**Int** = 60
3. }
4. **class** Bike **extends** Vehicle{
5. override **val** speed:**Int** = 100
6. **def** show(){
7. println(speed)
8. }
9. }
11. **object** MainObject{
12. **def** main(args:Array[String]){
13. **var** b = **new** Bike()
14. b.show()
15. }
16. }

Output:

Error - value speed cannot override final member

Scala Final Method

Final method declare in the parent class can't be override. You can make any method to final if you don't want to get it overridden. Attempt to override final method will cause to a compile time error.

Scala Final Method Example

1. **class** Vehicle{
2. **final** **def** show(){
3. println("vehicle is running")
4. }
5. }
6. **class** Bike **extends** Vehicle{
7. //override val speed:Int = 100
8. override **def** show(){
9. println("bike is running")
10. }
11. }
12. **object** MainObject{
13. **def** main(args:Array[String]){
14. **var** b = **new** Bike()
15. b.show()
16. }
17. }

Output:

method show cannot override final member

override def show(){

^

one error found

Scala Final Class Example

You can also make final class. Final class can't be inherited. If you make a class final, it can't be extended further.

1. **final** **class** Vehicle{
2. **def** show(){
3. println("vehicle is running")
4. }
6. }
8. **class** Bike **extends** Vehicle{
9. override **def** show(){
10. println("bike is running")
11. }
12. }
14. **object** MainObject{
15. **def** main(args:Array[String]){
16. **var** b = **new** Bike()
17. b.show()
18. }
19. }

Output:

error: illegal inheritance from final class Vehicle

class Bike extends Vehicle{

^

one error found

Scala Abstract Class

A class which is declared with abstract keyword is known as abstract class. An abstract class can have abstract methods and non-abstract methods as well. Abstract class is used to achieve abstraction. Abstraction is a process in which we hide complex implementation details and show only functionality to the user.

In scala, we can achieve abstraction by using abstract class and trait. We have discussed about these in detail here.

Scala Abstract Class Example

In this example, we have created a Bike abstract class. It contains an abstract method. A class Hero extends it and provides implementation of its run method.

A class that extends an abstract class must provide implementation of its all abstract methods. You can't create object of an abstract class.

1. abstract **class** Bike{
2. **def** run()
3. }
5. **class** Hero **extends** Bike{
6. **def** run(){
7. println("running fine...")
8. }
9. }
11. **object** MainObject{
12. **def** main(args: Array[String]){
13. **var** h = **new** Hero()
14. h.run()
15. }
16. }

Output:

running fine...

Scala Abstract Class Example: Having Constructor, Variables and Abstract Methods

1. abstract **class** Bike(a:**Int**){             // Creating constructor
2. **var** b:**Int** = 20                      // Creating variables
3. **var** c:**Int** = 25
4. **def** run()                           // Abstract method
5. **def** performance(){                  // Non-abstract method
6. println("Performance awesome")
7. }
8. }
10. **class** Hero(a:**Int**) **extends** Bike(a){
11. c = 30
12. **def** run(){
13. println("Running fine...")
14. println("a = "+a)
15. println("b = "+b)
16. println("c = "+c)
17. }
18. }
20. **object** MainObject{
21. **def** main(args: Array[String]){
22. **var** h = **new** Hero(10)
23. h.run()
24. h.performance()
25. }
26. }

Output:

Running fine...

a = 10

b = 20

c = 30

Performance awesome

Scala Abstract Class Example: Abstract Method is not implemented

In this example, we didn't implement abstract method run(). Compiler reports an error during compilation of this program. Error message is given below in output section.

1. abstract **class** Bike{
2. **def** run()               // Abstract method
3. }
5. **class** Hero **extends** Bike{        // Not implemented in this class
6. **def** runHero(){
7. println("Running fine...")
8. }
9. }
11. **object** MainObject{
12. **def** main(args: Array[String]){
13. **var** h = **new** Hero()
14. h.runHero()
15. }
16. }

Output:

error: class Hero needs to be abstract, since method run in class Bike of type ()Unit is not defined

class Hero extends Bike{

^

one error found

To avoid this problem either you must implement all abstract members of abstract class or make your class abstract too.

# Scala Trait

A trait is like an interface with a partial implementation. In scala, trait is a collection of abstract and non-abstract methods. You can create trait that can have all abstract methods or some abstract and some non-abstract methods.

A variable that is declared either by using val or var keyword in a trait get internally implemented in the class that implements the trait. Any variable which is declared by using val or var but not initialized is considered abstract.

Traits are compiled into Java interfaces with corresponding implementation classes that hold any methods implemented in the traits.

## Scala Trait Example

1. **trait** Printable{
2. **def** print()
3. }
5. **class** A4 **extends** Printable{
6. **def** print(){
7. println("Hello")
8. }
9. }
11. **object** MainObject{
12. **def** main(args:Array[String]){
13. **var** a = **new** A4()
14. a.print()
15. }
16. }

Output:

Hello

If a class extends a trait but does not implement the members declared in that trait, it must be declared abstract. Let's see an example.

## Scala Trait Example

1. **trait** Printable{
2. **def** print()
3. }
5. abstract **class** A4 **extends** Printable{            // Must declared as abstract class
6. **def** printA4(){
7. println("Hello, this is A4 Sheet")
8. }
9. }

## Scala Trait Example: Implementing Multiple Traits in a Class

If a class implements multiple traits, it will extend the first trait, class, abstract class. with keyword is used to extend rest of the traits.

You can achieve multiple inheritances by using trait.

1. **trait** Printable{
2. **def** print()
3. }
5. **trait** Showable{
6. **def** show()
7. }
9. **class** A6 **extends** Printable **with** Showable{
10. **def** print(){
11. println("This is printable")
12. }
13. **def** show(){
14. println("This is showable");
15. }
16. }
18. **object** MainObject{
19. **def** main(args:Array[String]){
20. **var** a = **new** A6()
21. a.print()
22. a.show()
23. }
24. }

Output:

This is printable

This is showable

## Scala Trait having abstract and non-abstract methods

You can also define method in trait as like in abstract class. I.e. you can treat trait as abstract class also. In scala, trait is almost same as abstract class except that it can't have constructor. You can't extend multiple abstract classes but can extend multiple traits.

### **Scala Trait Example**

1. **trait** Printable{
2. **def** print()         // Abstract method
3. **def** show(){         // Non-abstract method
4. println("This is show method")
5. }
6. }
8. **class** A6 **extends** Printable{
9. **def** print(){
10. println("This is print method")
11. }
12. }
14. **object** MainObject{
15. **def** main(args:Array[String]){
16. **var** a = **new** A6()
17. a.print()
18. a.show()
19. }
20. }

Output:

This is print method

This is show method

Scala Trait Mixins

In scala, trait mixins means you can extend any number of traits with a class or abstract class. You can extend only traits or combination of traits and class or traits and abstract class.

It is necessary to maintain order of mixins otherwise compiler throws an error.

You can use mixins in scala like this:

Scala Trait Example: Mixins Order Not Maintained

In this example, we have extended a trait and an abstract class. Let's see what happen.

1. **trait** Print{
2. **def** print()
3. }
5. abstract **class** PrintA4{
6. **def** printA4()
7. }
9. **class** A6 **extends** Print **with** PrintA4{
10. **def** print(){                // Trait print
11. println("print sheet")
12. }
13. **def** printA4(){              // Abstract class printA4
14. println("Print A4 Sheet")
15. }
16. }
18. **object** MainObject{
19. **def** main(args:Array[String]){
20. **var** a = **new** A6()
21. a.print()
22. a.printA4()
23. }
24. }

Output:

error: class PrintA4 needs to be a trait to be mixed in

class A6 extends Print with PrintA4{

^

one error found

The above program throws a compile time error, because we didn't maintain mixins order.

Scala Mixins Order

The right mixins order of trait is that any class or abstract class which you want to extend, first extend this. All the traits will be extended after this class or abstract class.

Scala Trait Example: Mixins Order Maintained

1. **trait** Print{
2. **def** print()
3. }
5. abstract **class** PrintA4{
6. **def** printA4()
7. }
9. **class** A6 **extends** PrintA4 **with** Print{            // First one is abstract class second one is trait
10. **def** print(){                                        // Trait print
11. println("print sheet")
12. }
13. **def** printA4(){                                      // Abstract class printA4
14. println("Print A4 Sheet")
15. }
16. }
18. **object** MainObject{
19. **def** main(args:Array[String]){
20. **var** a = **new** A6()
21. a.print()
22. a.printA4()
23. }
24. }

Output:

print sheet

Print A4 Sheet

Another Example of Scala Trait

Here, we have used one more approach to extend trait in our program. In this approach, we extend trait during object creation. Let's see an example.

1. **trait** Print{
2. **def** print()
3. }
5. abstract **class** PrintA4{
6. **def** printA4()
7. }
9. **class** A6 **extends** PrintA4 {
10. **def** print(){                             // Trait print
11. println("print sheet")
12. }
13. **def** printA4(){                              // Abstract class printA4
14. println("Print A4 Sheet")
15. }
16. }
18. **object** MainObject{
19. **def** main(args:Array[String]){
20. **var** a = **new** A6() **with** Print             // You can also extend trait during object creation
21. a.print()
22. a.printA4()
23. }
24. }

Output:

print sheet

Print A4 Sheet

Scala Access Modifier

Access modifier is used to define accessibility of data and our code to the outside world. You can apply accessibly to classes, traits, data members, member methods and constructors etc. Scala provides least accessibility to access to all. You can apply any access modifier to your code according to your application requirement.

Scala provides only three types of access modifiers, which are given below:

1. No modifier
2. Protected
3. Private

In scala, if you don't mention any access modifier, it is treated as no modifier.

Following table contains information about accessbilty of access modifiers.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Modifier** | **Outside package** | **Package** | **Class** | **Subclass** | **Companion** |
| No access modifier | Yes | Yes | Yes | Yes | Yes |
| Protected | No | No | Yes | Yes | Yes |
| Private | No | No | Yes | No | Yes |

Scala Example: Private Access Modifier

In scala, private access modifier is used to make data accessible only within class in which it is declared. It is most restricted and keeps your data in limited scope. Private data members does not inherit into subclasses.

1. **class** AccessExample{
2. **private** **var** a:**Int** = 10
3. **def** show(){
4. println(a)
5. }
6. }
7. **object** MainObject{
8. **def** main(args:Array[String]){
9. **var** p = **new** AccessExample()
10. p.a = 12
11. p.show()
12. }
13. }

Output:

error: variable a in class AccessExample cannot be accessed in AccessExample

p.a = 12

^

one error found

Scala Example: Protected Access Modifier

Protected access modifier is accessible only within class, sub class and companion object. Data members declared as protected are inherited in subclass. Let's see an example.

1. **class** AccessExample{
2. **protected** **var** a:**Int** = 10
3. }
4. **class** SubClass **extends** AccessExample{
5. **def** display(){
6. println("a = "+a)
7. }
8. }
9. **object** MainObject{
10. **def** main(args:Array[String]){
11. **var** s = **new** SubClass()
12. s.display()
13. }
14. }

Output:

a = 10

Scala Example: No-Access-Modifier

In scala, when you don't mention any access modifier, it is treated as no-access-modifier. It is same as public in java. It is least restricted and can easily accessible from anywhere inside or outside the package.

1. **class** AccessExample{
2. **var** a:**Int** = 10
3. **def** show(){
4. println(" a = "+a)
5. }
6. }
8. **object** MainObject{
9. **def** main(args:Array[String]){
10. **var** a = **new** AccessExample()
11. a.show()
12. }
13. }

Output:

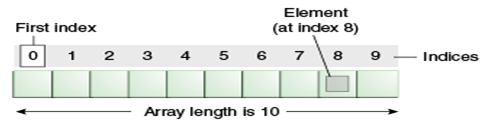
a = 10

# 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. }

Output:

1

2

3

4

5

Scala Multidimensional Array

Multidimensional array is an array which store data in matrix form. You can create from two dimensional to three, four and many more dimensional array according to your need. Below we have mentioned array syntax. Scala provides an ofDim method to create multidimensional array.

Multidimensional Array Syntax

1. **var** arrayName = Array.ofDim[ArrayType](NoOfRows,NoOfColumns) or
2. **var** arrayName = Array(Array(element?), Array(element?), ?)

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. }

Output:

0 0

15 0

Third Element = 0

Scala Multidimensional Array by using Array of Array

Apart from ofDim you can also create multidimensional array by using array of array. In this example, we have created multidimensional array by using array of array.

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

Output:

1 2 3 4 5

6 7 8 9 10

Scala Addition of Two Matrix Example

You can manipulate array elements in scala. Here, we are adding two array elements and storing result into third array.

1. **class** ArrayExample{
2. **var** arr1 = Array(Array(1,2,3,4,5), Array(6,7,8,9,10))   // Creating multidimensional array
3. **var** arr2 = Array(Array(1,2,3,4,5), Array(6,7,8,9,10))
4. **var** arr3 = Array.ofDim[**Int**](2,5)
5. **def** show(){
6. **for**(i<- 0 **to** 1){               // Traversing elements using loop
7. **for**(j<- 0 **to** 4){
8. arr3(i)(j) = arr1(i)(j)+arr2(i)(j)
9. print(" "+arr3(i)(j))
10. }
11. println()
12. }
13. }
14. }
16. **object** MainObject{
17. **def** main(args:Array[String]){
18. **var** a = **new** ArrayExample()
19. a.show()
20. }
21. }

Output:

2 4 6 8 10

12 14 16 18 20

Scala String

In scala, string is a combination of characters or we can say it is a sequence of characters. It is index based data structure and use linear approach to store data into memory. String is immutable in scala like java. You can manipulate string and can apply method to get desire result but you can't change original string object.

Scala String Example

This is a simple string example in which we have created a string variable.

1. **class** StringExample{
2. **var** s1 = "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:

Scala string example

Scala Immutable String Example 1

In this example, we are creating a string variable and trying to modify string object. Let's see what happen.

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

Output:

Scala string example

Scala Immutable String Example 2

In this example, we are modifying the string and assigning it to a variable so that we can keep reference of modified string. Here, a new string object is created after modification. The s1 variable now refers to new string object. The old string object is still available in the memory but does not have any reference.

1. **class** StringExample{
2. **var** s1 = "Scala string example"
3. s1 = "This is "+s1
4. **def** show(){
5. println(s1)
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 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. }

Output:

false

true

Scala String Methods

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 String Interpolation

Starting in Scala 2.10.0, Scala offers a new mechanism to create strings from your data. It is called string interpolation. String interpolation allows users to embed variable references directly in processed string literals. Scala provides three string interpolation methods: s, f and raw.

Scala Program Example: Without using s Method

This is simple example which does not use s method in string.

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

Output:

value of pi = 3.14

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. }

Output:

value of pi = 3.14

Scala String Interpolation Example By using s Method

The s method of string interpolation allows us to pass variable in string object. You don't need to use + operator to format your output string. In the following example, a string variable is passed to string in the print function. This variable is evaluated by compiler and variable is replaced by value.

1. **class** StringExample{
2. **var** s1 = "Scala string example"
3. **def** show(){
4. println(s"This is $s1")
5. }
6. }

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 String Interpolation Example By using f Method

The f method is used to format your string output. It is like printf function of c language which is used to produce formatted output. You can pass your variables of any type in the print function.

1. **class** StringExample{
2. **var** s1 = "Scala string example"
3. **var** version = 2.12
4. **def** show(){
5. println(f"This is $s1%s, scala version is $version%2.2f")
6. }
7. }

10. **object** MainObject{
11. **def** main(args:Array[String]){
12. **var** s = **new** StringExample()
13. s.show()
14. }
15. }

Output:

This is Scala string example, scala version is 2.12

Scala String Interpolation Example By using raw Method

The raw method of string interpolation is used to produce raw string. It does not interpret special char present in the string. Let's see an example.

1. **class** StringExample{
2. **var** s1 = "Scala \tstring \nexample"
3. **var** s2 = raw"Scala \tstring \nexample"
4. **def** show(){
5. println(s1)
6. println(s2)
7. }
8. }
10. **object** MainObject{
11. **def** main(args:Array[String]){
12. **var** s = **new** StringExample()
13. s.show()
14. }
15. }

Output:

Scala string

example

Scala \nstring \nexample

Scala Exception Handling

Exception handling is a mechanism which is used to handle abnormal conditions. You can also avoid termination of your program unexpectedly.

Scala makes "checked vs unchecked" very simple. It doesn't have checked exceptions. All exceptions are unchecked in Scala, even SQLException and IOException.

Scala Program Example without Exception Handling

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

Output:

java.lang.ArithmeticException: / by zero

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. }

Output:

found a unknown exceptionjava.lang.ArrayIndexOutOfBoundsException: 10

Rest of the code is executing...

cala Finally

The finally block is used to release resources during exception. Resources may be file, network connection, database connection etc. the finally block executes guaranteed. The following program illustrate the use of finally block.

Scala Finally Block Example

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: Exception =>println(ex)
10. **case** th: Throwable=>println("found a unknown exception"+th)
11. }
12. finally{
13. println("Finaly block always executes")
14. }
15. println("Rest of the code is executing...")
16. }
17. }

20. **object** MainObject{
21. **def** main(args:Array[String]){
22. **var** e = **new** ExceptionExample()
23. e.divide(100,10)
25. }
26. }

Output:

java.lang.ArrayIndexOutOfBoundsException: 10

Finally block always executes

Rest of the code is executing...

Scala Throw keyword

You can throw exception explicitly in you code. Scala provides throw keyword to throw exception. The throw keyword mainly used to throw custom exception. An example is given below of using scala throw exception keyword.

Scala Throw Example

1. **class** ExceptionExample2{
2. **def** validate(age:**Int**)={
3. **if**(age<18)
4. **throw** **new** ArithmeticException("You are not eligible")
5. **else** println("You are eligible")
6. }
7. }
9. **object** MainObject{
10. **def** main(args:Array[String]){
11. **var** e = **new** ExceptionExample2()
12. e.validate(10)
14. }
15. }

Output:

java.lang.ArithmeticException: You are not eligible

Scala Throws Keyword

Scala provides throws keyword to declare exception. You can declare exception with method definition. It provides information to the caller function that this method may throw this exception. It helps to caller function to handle and enclose that code in try-catch block to avoid abnormal termination of program. In scala, you can either use throws keyword or throws annotation to declare exception.

Scala Throws Example

1. **class** ExceptionExample4{
2. @throws(classOf[NumberFormatException])
3. **def** validate()={
4. "abc".toInt
5. }
6. }
8. **object** MainObject{
9. **def** main(args:Array[String]){
10. **var** e = **new** ExceptionExample4()
11. **try**{
12. e.validate()
13. }**catch**{
14. **case** ex : NumberFormatException => println("Exception handeled here")
15. }
16. println("Rest of the code executing...")
17. }
18. }

Output:

Exception handeled here

Rest of the code executing...

Scala Custom Exception

In scala, you can create your own exception. It is also known as custom exceptions. You must extend Exception class while declaring custom exception class. You can create your own exception message in custom class. Let's see an example.

Scala Custom Exception Example

1. **class** InvalidAgeException(s:String) **extends** Exception(s){}
2. **class** ExceptionExample{
3. @throws(classOf[InvalidAgeException])
4. **def** validate(age:**Int**){
5. **if**(age<18){
6. **throw** **new** InvalidAgeException("Not eligible")
7. }**else**{
8. println("You are eligible")
9. }
10. }
11. }
12. **object** MainObject{
13. **def** main(args:Array[String]){
14. **var** e = **new** ExceptionExample()
15. **try**{
16. e.validate(5)
17. }**catch**{
18. **case** e : Exception => println("Exception Occured : "+e)
19. }
20. }
21. }

Output:

Exception Occured : InvalidAgeException: Not eligible

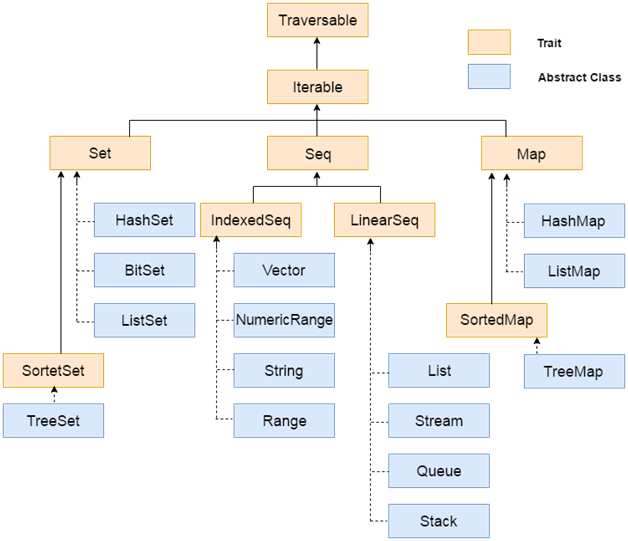
# Scala Collection

Scala provides rich set of collection library. It contains classes and traits to collect data. These collections can be mutable or immutable. You can use them according to your requirement. **Scala.collection.mutable** package contains all the mutable collections. You can add, remove and update data while using this package.

**Scala.collection.immutable** contains all the immutable collections. It does not allow you to modify data. Scala imports this package by default. If you want mutable collection, you must import **scala.collection.mutable** package in your code.

## Scala Immutable Collections Hierarchy

The scala.collection.immutable package contains all the immutable abstract classes and traits for collections.



### **Scala Traversable**

It is a trait and used to traverse collection elements. It is a base trait for all scala collections.

It implements the methods which are common to all collections.

### **Some Significant Methods of Traversable Trait**

|  |  |
| --- | --- |
| **Method** | **Description** |
| def head: A | It returns the first element of collection. |
| def init: Traversable[A] | It returns all elements except last one. |
| def isEmpty: Boolean | It checks whether the collection is empty or not. It returns either true or false. |
| def last: A | It returns the last element of this collection. |
| def max: A | It returns the largest element of this collection. |
| def min: A | It returns smallest element of this collection |
| def size: Int | It is used to get size of this traversable and returns a number of elements present in this traversable. |
| def sum: A | It returns sum of all elements of this collection. |
| def tail: Traversable[A] | It returns all elements except first. |
| def toArray: Array[A] | It converts this collection to an array. |
| def toList: List[A] | It converts this collection to a list. |
| def toSeq: Seq[A] | It converts this collection to a sequence. |
| def toSet[B >: A]: immutable.Set[B] | It converts this collection to a set. |

## Scala Iterable

It is a next trait from the top of the hierarchy and a base trait for iterable collections. It extends traversable trait and provides important methods to concrete classes.

# Scala Set

It is used to store unique elements in the set. It does not maintain any order for storing elements. You can apply various operations on them. It is defined in the Scala.collection.immutable package.

### **Scala Set Syntax**

1. **val** variableName:Set[Type] = Set(element1, element2,... elementN) or
2. **val** variableName = Set(element1, element2,... elementN)

### **Scala Set Example**

In this example, we have created a set. You can create an empty set also. Let's see how to create a set.

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **val** set1 = Set()                            // An empty set
5. **val** games = Set("Cricket","Football","Hocky","Golf")    // Creating a set with elements
6. println(set1)
7. println(games)
8. }
9. }

Output:

Set() // an empty set

Set(Cricket,Football,Hocky,Golf)

### **Scala Set Example 2**

In Scala, Set provides some predefined properties to get information about set. You can get first or last element of Set and many more. Let's see an example.

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **val** games = Set("Cricket","Football","Hocky","Golf")
5. println(games.head)             // Returns first element present in the set
6. println(games.tail)         // Returns all elements except first element.
7. println(games.isEmpty)          // Returns either true or false
8. }
9. }

Output:

Cricket

Set(Football, Hocky, Golf)

false

### **Scala Set Example: Merge two Set**

You can merge two sets into a single set. Scala provides a predefined method to merge sets. In this example, ++ method is used to merge two sets.

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **val** games = Set("Cricket","Football","Hocky","Golf")
5. **val** alphabet = Set("A","B","C","D","E")
6. **val** mergeSet = games ++ alphabet            // Merging two sets
7. println("Elements in games set: "+games.size)   // Return size of collection
8. println("Elements in alphabet set: "+alphabet.size)
9. println("Elements in mergeSet: "+mergeSet.size)
10. println(mergeSet)
11. }
12. }

Output:

Elements in games set: 4

Elements in alphabet set: 5

Elements in mergeSet: 9

Set(E, Football, Golf, Hocky, A, B, C, Cricket, D)

This example also proves that the merge set does not maintain order to store elements.

### **Scala Set Example 2**

You can check whether element is present in the set or not. The following example describe the use of contains() method.

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **val** games = Set("Cricket","Football","Hocky","Golf")
5. println(games)
6. println("Elements in set: "+games.size)
7. println("Golf exists in the set : "+games.contains("Golf"))
8. println("Racing exists in the set : "+games.contains("Racing"))
10. }
11. }

Output:

Set(Cricket, Football, Hocky, Golf)

Elements in set: 4

Golf exists in the set : true

Racing exists in the set : false

### **Scala Set Example: Adding and Removing Elements**

You can add or remove elements from the set. You can add only when your code is mutable. In this example, we are adding and removing elements of the set.

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** games = Set("Cricket","Football","Hocky","Golf")
5. println(games)
6. games += "Racing"               // Adding new element
7. println(games)
8. games += "Cricket"              // Adding new element, it does not allow duplicacy.
9. println(games)
10. games -= "Golf"             // Removing element
11. println(games)
12. }
13. }

Output:

Set(Cricket, Football, Hocky, Golf)

Set(Football, Golf, Hocky, Cricket, Racing)

Set(Football, Golf, Hocky, Cricket, Racing)

Set(Football, Hocky, Cricket, Racing)

### **Scala Set Example: Iterating Set Elements using for loop**

You can iterate set elements either by using for loop or foreach loop. You can also filter elements during iteration. In this example have used for loop to iterate set elements.

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** games = Set("Cricket","Football","Hocky","Golf")
5. **for**(game <- games){
6. println(game)
7. }
8. }
9. }

Output:

Cricket

Football

Hocky

Golf

### **Scala Set Example Iterating Elements using foreach loop**

In this example, we are using foreach loop to iterate set elements.

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** games = Set("Cricket","Football","Hocky","Golf")
5. games.foreach((element:String)=> println(element))
6. }
7. }

Output:

Cricket

Football

Hocky

Golf

### **Scala Set Example: Set Operations**

In scala Set, you can also use typical math operations like: intersection and union. In the following example we have used predefined methods to perform set operations.

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** games = Set("Cricket","Football","Hocky","Golf","C")
5. **var** alphabet = Set("A","B","C","D","E","Golf")
6. **var** setIntersection = games.intersect(alphabet)
7. println("Intersection by using intersect method: "+setIntersection)
8. println("Intersection by using & operator: "+(games & alphabet))
9. **var** setUnion = games.union(alphabet)
10. println(setUnion)
11. }
12. }

Output:

Intersection by using intersect method: Set(Golf, C)

Intersection by using & operator: Set(Golf, C)

Set(E, Football, Golf, Hocky, A, B, C, Cricket, D)

### **Scala SortedSet**

In scala, SortedSet extends Set trait and provides sorted set elements. It is useful when you want sorted elements in the Set collection. You can sort integer values and string as well.

It is a trait and you can apply all the methods defined in the traversable trait and Set trait.

### **Scala SortedSet Example**

In the following example, we have used SortedSet to store integer elements. It returns a Set after sorting elements.

1. **import** scala.collection.immutable.SortedSet
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** numbers: SortedSet[**Int**] = SortedSet(5,8,1,2,9,6,4,7,2)
5. numbers.foreach((element:**Int**)=> println(element))
6. }
7. }

Output:

1

2

4

5

6

7

8

9

Scala HashSet Example

In the following example, we have created a HashSet to store elements. Here, foreach is used to iterate elements.

1. **import** scala.collection.immutable.HashSet
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** hashset = HashSet(4,2,8,0,6,3,45)
5. hashset.foreach((element:**Int**) => println(element+" "))
6. }
7. }

Output:

0

6

2

45

3

8

4

# Scala BitSet

Bitsets are sets of non-negative integers which are represented as variable-size arrays of bits packed into 64-bit words. The memory footprint of a bitset is determined by the largest number stored in it. It extends Set trait.

## Scala BitSet Example

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** numbers = BitSet(1,5,8,6,9,0)
5. numbers.foreach((element:**Int**) => println(element))
6. }
7. }

Output:

0

1

5

6

8

9

### **Scala BitSet Example: Adding and Removing Elements**

You can perform basic operations like adding and deleting in the bitset. In the following example, we have applied these operations.

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** numbers = BitSet(1,5,8,6,9,0)
5. numbers.foreach((element:**Int**) => print(element+" "))
6. numbers += 20               // Adding an element
7. print("\nAfter adding 20: ")
8. numbers.foreach((element:**Int**) => print(element+" "))
9. numbers-=0              // Deleting an element
10. print("\nAfter deleting 0: ")
11. numbers.foreach((element:**Int**) => print(element+" "))
12. }
13. }

Output:

0 1 5 6 8 9

After adding 20: 0 1 5 6 8 9 20

After deleting 0: 1 5 6 8 9 20

# Scala ListSet

In scala, ListSet class implements immutable sets using a list-based data structure. Elements are stored internally in reversed insertion order, which means the newest element is at the head of the list. It maintains insertion order.

This collection is suitable only for a small number of elements. You can create empty ListSet either by calling the constructor or by applying the function ListSet.empty. Its iterate and traversal methods visit elements in the same order in which they were first inserted.

## Scala ListSet Example

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** listset = ListSet(4,2,8,0,6,3,45)
5. listset.foreach((element:**Int**) => println(element+" "))
6. }
7. }

Output:

4

2

8

0

6

3

45

### **Scala ListSet Example: Creating ListSet and Adding Elements**

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** listset:ListSet[String] = **new** ListSet()                 // Creating empty ListSet by using constructor
5. **var** listset2:ListSet[String] = ListSet.empty                // Creating an empty listset
6. println("listset: "+listset)
7. println("listset2: "+listset2)
8. println("After adding new elements:")
9. listset+="India"            // Adding new element
10. listset2+="Russia"          // Adding new element
11. println("listset: "+listset)
12. println("listset2: "+listset2)
13. }
14. }

Output:

listset: ListSet()

listset2: ListSet()

After adding new elements:

listset: ListSet(India)

listset2: ListSet(Russia)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**next →**](http://www.javatpoint.com/scala-vector)[**← prev**](http://www.javatpoint.com/scala-listset) Scala Seq Seq is a trait which represents indexed sequences that are guaranteed immutable. You can access elements by using their indexes. It maintains insertion order of elements.  Sequences support a number of methods to find occurrences of elements or subsequences. It returns a list. Scala Seq Example In the following example, we are creating Seq and accessing elements from Seq.   1. **import** scala.collection.immutable.\_ 2. **object** MainObject{ 3. **def** main(args:Array[String]){ 4. **var** seq:Seq[**Int**] = Seq(52,85,1,8,3,2,7) 5. seq.foreach((element:**Int**) => print(element+" ")) 6. println("\nAccessing element by using index") 7. println(seq(2)) 8. } 9. }   Output:  52 85 1 8 3 2 7  Accessing element by using index  1  You can also access elements in reverse order by using reverse method. Below we have listed some commonly used method and their description. **Commonly used Methods of Seq**  |  |  | | --- | --- | | **Method** | **Description** | | def contains[A1 >: A](elem: A1): Boolean | Check whether the given element present in this sequence. | | def copyToArray(xs: Array[A], start: Int, len: Int): Unit | It copies the seq elements to an array. | | def endsWith[B](that: GenSeq[B]): Boolean | It tests whether this sequence ends with the given sequence or not. | | def head: A | It selects the first element of this seq collection. | | def indexOf(elem: A): Int | It finds index of first occurrence of a value in this immutable sequence. | | def isEmpty: Boolean | It tests whether this sequence is empty or not. | | def lastIndexOf(elem: A): Int | It finds index of last occurrence of a value in this immutable sequence. | | def reverse: Seq[A] | It returns new sequence with elements in reversed order. |  Scala Seq Example In this example, we have applied some predefined methods of Seq trait.   1. **import** scala.collection.immutable.\_ 2. **object** MainObject{ 3. **def** main(args:Array[String]){ 4. **var** seq:Seq[**Int**] = Seq(52,85,1,8,3,2,7) 5. seq.foreach((element:**Int**) => print(element+" ")) 6. println("\nis Empty: "+seq.isEmpty) 7. println("Ends with (2,7): "+ seq.endsWith(Seq(2,7))) 8. println("contains 8: "+ seq.contains(8)) 9. println("last index of 3 : "+seq.lastIndexOf(3)) 10. println("Reverse order of sequence: "+seq.reverse) 11. } 12. }   Output:  52 85 1 8 3 2 7  is Empty: false  Ends with (2,7): true  contains 8: true  last index of 3 : 4  Reverse order of sequence: List(7, 2, 3, 8, 1, 85, 52) |

Scala Vector

Vector is a general-purpose, immutable data structure. It provides random access of elements. It is good for large collection of elements.

It extends an abstract class AbstractSeq and IndexedSeq trait.

Scala Vector Example

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** vector:Vector[**Int**] = Vector(5,8,3,6,9,4) //Or
5. **var** vector2 = Vector(5,2,6,3)
6. **var** vector3 = Vector.empty
7. println(vector)
8. println(vector2)
9. println(vector3)
10. }
11. }

Output:

Vector(5, 8, 3, 6, 9, 4)

Vector(5, 2, 6, 3)

Vector(

Scala Vector Example

In the following example, we have created a vector. You can also add new element and merge two vectors.

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** vector = Vector("Hocky","Cricket","Golf")
5. **var** vector2 = Vector("Swimming")
6. print("Vector Elements: ")
7. vector.foreach((element:String) => print(element+" "))
8. **var** newVector  = vector :+ "Racing"                             // Adding a new element into vector
9. print("\nVector Elements after adding: ")
10. newVector.foreach((element:String) => print(element+" "))
11. **var** mergeTwoVector = newVector ++ vector2                       // Merging two vector
12. print("\nVector Elements after merging: ")
13. mergeTwoVector.foreach((element:String) => print(element+" "))
14. **var** reverse = mergeTwoVector.reverse                            // Reversing vector elements
15. print("\nVector Elements after reversing: ")
16. reverse.foreach((element:String) => print(element+" "))
17. **var** sortedVector = mergeTwoVector.sorted                        // Sorting vector elements
18. print("\nVector Elements after sorting: ")
19. sortedVector.foreach((element:String) => print(element+" "))
20. }
21. }

Output:

Vector Elements: Hocky Cricket Golf

Vector Elements after adding: Hocky Cricket Golf Racing

Vector Elements after merging: Hocky Cricket Golf Racing Swimming

Vector Elements after reversing: Swimming Racing Golf Cricket Hocky

Vector Elements after sorting: Cricket Golf Hocky Racing Swimming

Scala List

List is used to store ordered elements. It extends LinearSeq trait. It is a class for immutable linked lists. This class is good for last-in-first-out (LIFO), stack-like access patterns.

It maintains order of elements and can contain duplicates elements also.

Scala List Example

In this example, we have created two lists. Here, both lists have different syntax to create list.

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** list = List(1,8,5,6,9,58,23,15,4)
5. **var** list2:List[**Int**] = List(1,8,5,6,9,58,23,15,4)
6. println(list)
7. println(list2)
8. }
9. }

Output:

List(1, 8, 5, 6, 9, 58, 23, 15, 4)

List(1, 8, 5, 6, 9, 58, 23, 15, 4)

Scala List Example: Applying Predefined Methods

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** list = List(1,8,5,6,9,58,23,15,4)
5. **var** list2 = List(88,100)
6. print("Elements: ")
7. list.foreach((element:**Int**) => print(element+" "))        // Iterating using foreach loop
8. print("\nElement at 2 index: "+list(2))             // Accessing element of 2 index
9. **var** list3 = list ++ list2                               // Merging two list
10. print("\nElement after merging list and list2: ")
11. list3.foreach((element:**Int**)=>print(element+" "))
12. **var** list4 = list3.sorted                            // Sorting list
13. print("\nElement after sorting list3: ")
14. list4.foreach((element:**Int**)=>print(element+" "))
15. **var** list5 = list3.reverse                           // Reversing list elements
16. print("\nElements in reverse order of list5: ")
17. list5.foreach((element:**Int**)=>print(element+" "))
19. }
20. }

Output:

Elements: 1 8 5 6 9 58 23 15 4

Element at 2 index: 5

Element after merging list and list2: 1 8 5 6 9 58 23 15 4 88 100

Element after sorting list3: 1 4 5 6 8 9 15 23 58 88 100

Elements in reverse order of list5: 100 88 4 15 23 58 9 6 5 8 1

# Scala Queue

Queue implements a data structure that allows inserting and retrieving elements in a first-in-first-out (FIFO) manner.

In scala, Queue is implemented as a pair of lists. One is used to insert the elements and second to contain deleted elements. Elements are added to the first list and removed from the second list.

## Scala Queue Example

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** queue = Queue(1,5,6,2,3,9,5,2,5)
5. **var** queue2:Queue[**Int**] = Queue(1,5,6,2,3,9,5,2,5)
6. println(queue)
7. println(queue2)
8. }
9. }

Output:

Queue(1, 5, 6, 2, 3, 9, 5, 2, 5)

Queue(1, 5, 6, 2, 3, 9, 5, 2, 5)

### **Scala Queue Example 2**

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** queue = Queue(1,5,6,2,3,9,5,2,5)
5. print("Queue Elements: ")
6. queue.foreach((element:**Int**)=>print(element+" "))
7. **var** firstElement = queue.front
8. print("\nFirst element in the queue: "+ firstElement)
9. **var** enqueueQueue = queue.enqueue(100)
10. print("\nElement added in the queue: ")
11. enqueueQueue.foreach((element:**Int**)=>print(element+" "))
12. **var** dequeueQueue = queue.dequeue
13. print("\nElement deleted from this queue: "+ dequeueQueue)
14. }
15. }

Output:

Queue Elements: 1 5 6 2 3 9 5 2 5

First element in the queue: 1

Element added in the queue: 1 5 6 2 3 9 5 2 5 100

Element deleted from this queue: (1,Queue(5, 6, 2, 3, 9, 5, 2, 5))

# Scala Stream

Stream is a lazy list. It evaluates elements only when they are required. This is a feature of scala. Scala supports lazy computation. It increases performance of your program.

## Scala Stream Example

In the following program, we have created a stream.

1. **object** MainObject{
2. **def** main(args:Array[String]){
3. **val** stream = 100 #:: 200 #:: 85 #:: Stream.empty
4. println(stream)
5. }
6. }

Output:

Stream(100, ?)

In the output, you can see that second element is not evaluated. Here, a question mark is displayed in place of element. Scala does not evaluate list until it is required.

### **Scala Stream Example: Applying Predefined Methods**

In the following example, we have used some predefined methods like toStream, which is used to iterate stream elements.

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** stream = 100 #:: 200 #:: 85 #:: Stream.empty
5. println(stream)
6. **var** stream2 = (1 **to** 10).toStream
7. println(stream2)
8. **var** firstElement = stream2.head
9. println(firstElement)
10. println(stream2.take(10))
11. println(stream.map{\_\*2})
12. }
13. }

Output:

Stream(100, ?)

Stream(1, ?)

1

Stream(1, ?)

Stream(200, ?)

# Scala Maps

Map is used to store elements. It stores elements in pairs of key and values. In scala, you can create map by using two ways either by using comma separated pairs or by using rocket operator.

## Scala maps Example

In the following example, we have both approaches to create map.

1. **object** MainObject{
2. **def** main(args:Array[String]){
3. **var** map = Map(("A","Apple"),("B","Ball"))
4. **var** map2 = Map("A"->"Aple","B"->"Ball")
5. **var** emptyMap:Map[String,String] = Map.empty[String,String]
6. println(map)
7. println(map2)
8. println("Empty Map: "+emptyMap)
9. }
10. }

Output:

Map(A -> Apple, B -> Ball)

Map(A -> Aple, B -> Ball)

Empty Map: Map()

### **Scala Map Example: Adding and Removing Elements**

You can add and remove new elements in maps. Scala provides you lots of predefined method. You can use them to perform operations on the Maps. In the following example, we have created a new Map.

1. **object** MainObject{
2. **def** main(args:Array[String]){
3. **var** map = Map("A"->"Apple","B"->"Ball")             // Creating map
4. println(map("A"))                            // Accessing value by using key
5. **var** newMap = map+("C"->"Cat")                  // Adding a new element to map
6. println(newMap)
7. **var** removeElement = newMap - ("B")                // Removing an element from map
8. println(removeElement)
9. }
10. }

Output:

Apple

Map(A -> Apple, B -> Ball, C -> Cat)

Map(A -> Apple, C -> Cat)

# Scala HashMap

HashMap is used to store element. It use hash code to store elements and return a map.

## HashMap Example

In this example, we have created a HashMap.

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** hashMap = **new** HashMap()
5. **var** hashMap2 = HashMap("A"->"Apple","B"->"Ball","C"->"Cat")
6. println(hashMap)
7. println(hashMap2)
8. }
9. }

Output:

Map()

Map(A -> Apple, B -> Ball, C -> Cat)

### **Scala HashMap Example: Adding and Accessing Elements**

In the following example, we have created a HashMap. this program add elements and access elements as well.

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** hashMap = HashMap("A"->"Apple","B"->"Ball","C"->"Cat")
5. hashMap.foreach {
6. **case** (key, value) => println (key + " -> " + value)       // Iterating elements
7. }
8. println(hashMap("B"))               // Accessing value by using key
9. **var** newHashMap = hashMap+("D"->"Doll")
10. newHashMap.foreach {
11. **case** (key, value) => println (key + " -> " + value)
12. }
14. }
15. }

Output:

A -> Apple

B -> Ball

C -> Cat

Ball

A -> Apple

B -> Ball

C -> Cat

D -> Doll

# Scala ListMap

This class implements immutable maps by using a list-based data structure. It maintains insertion order and returns ListMap. This collection is suitable for small elements.

You can create empty ListMap either by calling its constructor or using ListMap.empty method.

## Scala ListMap Example

In this example, we have created an empty ListMap and non-empty ListMap as well.

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** listMap = ListMap("Rice"->"100","Wheat"->"50","Gram"->"500")    // Creating listmap with elements
5. **var** emptyListMap = **new** ListMap()            // Creating an empty list map
6. **var** emptyListMap2 = ListMap.empty           // Creating an empty list map
7. println(listMap)
8. println(emptyListMap)
9. println(emptyListMap2)
10. }
11. }

Output:

ListMap(Rice -> 100, Wheat -> 50, Gram -> 500)

ListMap()

ListMap()

### **Scala ListMap Example: Applying Basic Operations**

1. **import** scala.collection.immutable.\_
2. **object** MainObject{
3. **def** main(args:Array[String]){
4. **var** listMap = ListMap("Rice"->"100","Wheat"->"50","Gram"->"500")    // Creating listmap with elements
5. listMap.foreach{
6. **case**(key,value)=>println(key+"->"+value)
7. }
8. println(listMap("Gram"))
9. **var** newListMap = listMap+("Pulses"->"550")
10. newListMap.foreach {
11. **case** (key, value) => println (key + " -> " + value)
12. }
13. }
14. }

Output:

Rice->100

Wheat->50

Gram->500

500

Rice -> 100

Wheat -> 50

Gram -> 500

Pulses -> 550

# 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 Tuple Example: Function Return Multiple Values

You can return multiple values by using tuple. Function does not return multiple values but you can do this with the help of tuple. In the following example, we are describing this process.

1. **object** MainObject{
2. **def** main(args:Array[String]){
3. **var** tupleValues = tupleFunction()
4. println("Iterating values: ")
5. tupleValues.productIterator.foreach(println)    // Iterating tuple values using productIterator
6. }
7. **def** tupleFunction()={
8. **var** tuple = (1,2.5,"India")
9. tuple
10. }
11. }

Output:

Iterating values:

1

2.5

India

# Scala File handling

Scala provides predefined methods to deal with file. You can create, open, write and read file. Scala provides a complete package **scala.io** for file handling.

In this chapter, we will discuss all these file operations in detail.

## 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. }

Output:

Hello, This is scala file

Apart from these for all kind of file handling operations, you need to use java.io package to access file methods.

# Scala Multithreading

Multithreading is a process of executing multiple threads simultaneously. It allows you to perform multiple operations independently.

You can achieved multitasking by using Multithreading. Threads are lightweight sub-processes which occupy less memory. Multithreading are used to develop concurrent applications in Scala.

Scala does not provide any separate library for creating thread. If you are familiar with multithreading concept of Java, you will come to know that it is similar except the syntax of Scala language itself.

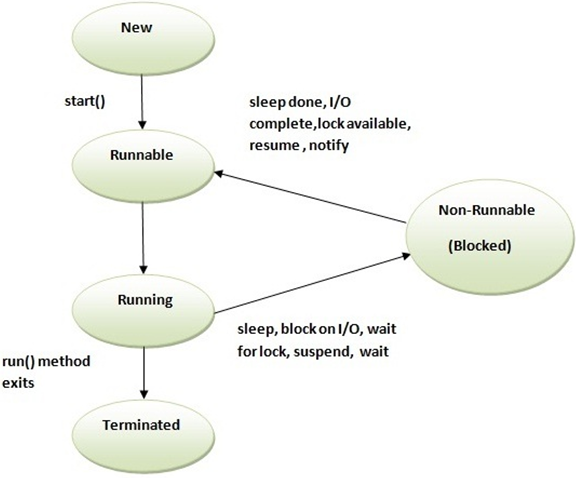
You can create thread either by extending Thread class or Runnable interface. Both provide a run method to provide specific implementation.

## Scala Thread Life Cycle

Thread life cycle is a span of time in which thread starts and terminates. It has various phases like new, runnable, terminate, block etc. Thread class provides various methods to monitor thread's states.

The Scala thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



### **1) New**

This is the first state of thread. It is just before starting of new thread.

### **2) Runnable**

This is the state when thread has been started but the thread scheduler has not selected it to be the running thread.

### **3) Running**

The thread is in running state if the thread scheduler has selected it.

### **4) Non-Runnable (Blocked)**

This is the state when the thread is still alive, but is currently not eligible to run due to waiting for input or resources.

### **5) Terminated**

A thread is in terminated or dead state when its run() method exits.

Scala Thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface

Scala Thread Example by Extending Thread Class

The following example extends Thread class and overrides run method. The start() method is used to start thread.

1. **class** ThreadExample **extends** Thread{
2. override **def** run(){
3. println("Thread is running?");
4. }
5. }
6. **object** MainObject{
7. **def** main(args:Array[String]){
8. **var** t = **new** ThreadExample()
9. t.start()
10. }
11. }

Output:

Thread is running...

Scala Thread Example by Extending Runnable Interface

The following example implements Runnable interface and overrides run method. The start() method is used to start thread.

1. **class** ThreadExample **extends** Runnable{
2. override **def** run(){
3. println("Thread is running...")
4. }
5. }
6. **object** MainObject{
7. **def** main(args:Array[String]){
8. **var** e = **new** ThreadExample()
9. **var** t = **new** Thread(e)
10. t.start()
11. }
12. }

Output:

Thread is running...

Scala Thread Methods

Thread class provides various methods to deals with thread's states. You can use these methods to control the flow of thread.

The following table contains commonly used methods of Thread class.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final String getName() | It returns thread's name. |
| public final int getPriority() | It returns thread's priority. |
| public Thread.State getState() | It returns the state of this thread. This method is designed for use in monitoring of the system state, not for synchronization control. |
| public final boolean isAlive() | It tests if this thread is alive. A thread is alive if it has been started and has not yet died. |
| public final void join() throws InterruptedException | It Waits for thread to die. |
| public void run() | If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns. |
| public final void setName(String name) | It is used to set thread name. |
| public final void setPriority(int newPriority) | It is used to set priority of a thread. |
| public static void sleep(long millis) throws InterruptedException | It is used to sleep executing thread for the specified number of milliseconds. |
| public static void yield() | It causes the currently executing thread object to temporarily pause and allow other threads to execute. |

Scala Thread sleep() Method

The sleep() method is used to sleep thread for the specified time. It takes time in milliseconds as an argument.

1. **class** ThreadExample **extends** Thread{
2. override **def** run(){
3. **for**(i<- 0 **to** 5){
4. println(i)
5. Thread.sleep(500)
6. }
7. }
9. }
11. **object** MainObject{
12. **def** main(args:Array[String]){
13. **var** t1 = **new** ThreadExample()
14. **var** t2 = **new** ThreadExample()
15. t1.start()
16. t2.start()
17. }
18. }

Output:

0

0

1

1

2

2

3

3

4

4

5

5

Scala Thread join() Method Example

The join() method waits for a thread to die. In other words, The join() method is used to hold the execution of currently running thread until the specified thread finished it's execution.

1. **class** ThreadExample **extends** Thread{
2. override **def** run(){
3. **for**(i<- 0 **to** 5){
4. println(i)
5. Thread.sleep(500)
6. }
7. }
9. }
10. **object** MainObject{
11. **def** main(args:Array[String]){
12. **var** t1 = **new** ThreadExample()
13. **var** t2 = **new** ThreadExample()
14. **var** t3 = **new** ThreadExample()
15. t1.start()
16. t1.join()
17. t2.start()
18. t3.start()
19. }
20. }

Output:

0

1

2

3

4

5

0

0

1

1

2

2

3

3

4

4

5

5

Scala setName() Method Example

In the following example, we are setting and getting names of threads.

1. **class** ThreadExample() **extends** Thread{
2. override **def** run(){
3. **for**(i<- 0 **to** 5){
4. println(this.getName()+" - "+i)
5. Thread.sleep(500)
6. }
7. }
8. }
10. **object** MainObject{
11. **def** main(args:Array[String]){
12. **var** t1 = **new** ThreadExample()
13. **var** t2 = **new** ThreadExample()
14. **var** t3 = **new** ThreadExample()
15. t1.setName("First Thread")
16. t2.setName("Second Thread")
17. t1.start()
18. t2.start()
19. }
20. }

Output:

First Thread - 0

Second Thread - 0

Second Thread - 1

First Thread - 1

Second Thread - 2

First Thread - 2

Second Thread - 3

First Thread - 3

Second Thread - 4

First Thread - 4

Second Thread - 5

First Thread - 5

Scala Thread Priority Example

You can set thread priority by using it's predefined method. The following example sets priority for the thread.

1. **class** ThreadExample() **extends** Thread{
2. override **def** run(){
3. **for**(i<- 0 **to** 5){
4. println(this.getName())
5. println(this.getPriority())
6. Thread.sleep(500)
7. }
8. }
9. }
10. **object** MainObject{
11. **def** main(args:Array[String]){
12. **var** t1 = **new** ThreadExample()
13. **var** t2 = **new** ThreadExample()
14. t1.setName("First Thread")
15. t2.setName("Second Thread")
16. t1.setPriority(Thread.MIN\_PRIORITY)
17. t2.setPriority(Thread.MAX\_PRIORITY)
18. t1.start()
19. t2.start()
20. }
21. }

Output:

First Thread

Second Thread

10

1

Second Thread

10

First Thread

1

Second Thread

10

First Thread

1

Second Thread

10

First Thread

1

Second Thread

10

First Thread

1

Second Thread

10

First Thread

1

Scala Thread Multitasking Example

The following example is running multiple tasks by using multiple threads. This example explains that how can we implement multitasking in Scala.

1. **class** ThreadExample() **extends** Thread{
2. override **def** run(){
3. **for**(i<- 0 **to** 5){
4. println(i)
5. Thread.sleep(500)
6. }
7. }
8. **def** task(){
9. **for**(i<- 0 **to** 5){
10. println(i)
11. Thread.sleep(200)
12. }
13. }
14. }
16. **object** MainObject{
17. **def** main(args:Array[String]){
18. **var** t1 = **new** ThreadExample()
19. t1.start()
20. t1.task()
21. }
22. }

Output:

0

0

1

2

1

3

4

2

5

3

4

5

# Akka Tutorial

Akka tutorial provides basic and advanced concepts of Akka. Our Akka tutorial is designed for beginners and professionals.

Akka is a open-source library or a toolkit written in Scala to create concurrent, distributed and fault-tolerant application.

Our Akka tutorial includes all topics of Akka API such as What is Actor, Actor life cycle, Actors communication, Props, Child Actor, Actor System etc.

# What is Akka

Akka is a open-source library or a toolkit. It is used to create concurrent, distributed and fault-tolerant application. You can integrate this library into any JVM(Java Virtual Machine) support language. In this tutorial, we will integrate this into Scala programming language.

Akka is written in Scala. It implements Actor Based Model. The Actor Model provides a higher level of abstraction for writing concurrent and distributed applications. It helps to developer to deals with explicit locking and thread management. Akka makes it easier to write correct concurrent and parallel application.

## Why named Akka?

It is the name of a beautiful Swedish mountain up in the northern part of Sweden called Laponia. The mountain is also sometimes called 'The Queen of Laponia'.

Akka is also the name of a goddess in the Sami (the native Swedish population) mythology. She is the goddess that stands for all the beauty and good in the world. The mountain can be seen as the symbol of this goddess.

The name Akka is a palindrome of letters A and K as in Actor Kernel.

[**next →**](http://www.javatpoint.com/akka-actorsystem)[**← prev**](http://www.javatpoint.com/akka-tutorial)

# Actor

An actor is an entity which communicates to other actor by message passing. Actor has it's own state and behavior. As in object-oriented programming everything is an object same like everything is an actor in actor-based system.

In other words, we can say that an actor is an object that encapsulates state and behavior.

## How to create an Actor?

You can create actor by extending Actor trait and implementing it's receive method. This method is called each time a message is received by the actor. The receive method does pattern matching on the received message and decide what to do.

Here, we have created a simple **"Hello Akka"** program.

### **Hello Akka Example**

1. **import** akka.actor.Actor;          // Importing actor trait
2. **import** akka.actor.ActorSystem;
3. **import** akka.actor.Props;
4. **class** HelloAkka **extends** Actor{    // Extending actor trait
5. **def** receive = {                 //  Receiving message
6. **case** msg:String => println(msg)
7. **case** \_ =>println("Unknown message")      // Default case
8. }
9. }
11. **object** Main{
12. **def** main(args:Array[String]){
13. **var** actorSystem = ActorSystem("ActorSystem");                       // Creating ActorSystem
14. **var** actor = actorSystem.actorOf(Props[HelloAkka],"HelloAkka")        //Creating actor
15. actor ! "Hello Akka"                                                // Sending messages by using !
16. actor ! 100.52
17. }
18. }

**Output:**

Hello Akka

Unknown message

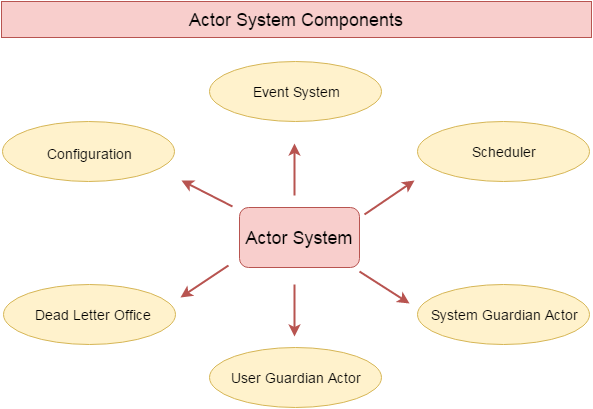
In the above example, we have created an actor "HelloAkka" by extending Actor trait and overriding receive method. You must provide match cases for all received messages. In case, if there is unknown message, you need to provide a default case as we did in above example.

# Akka ActorSystem

The ActorSystem is a root actor in actors structure. An ActorSystem is a hierarchical group of actors which share common configuration, e.g. dispatchers, deployments, remote capabilities and addresses. It is also the entry point for creating or looking up actors. It is an abstract class which extends to ActorRefFactory trait.

ActorSystem provides an actorOf() method which is used to create actor instance.

## Akka ActorSystem Components



### **1) Dead Letter Office**

Messages which cannot be delivered will be delivered to an actor called deadLetters. Messages sent via unreliable network will be lost without forwarding to dead letter office. The main use of this facility is for debugging purpose, especially if a message sent by an actor does not arrive consistently. You can implement this by importing **akka.actor.DeadLatter** package.

### **2) User Guardian Actor**

It is parent actor of actors created by user by using ActorSystem. This special guardian is used to achieve an orderly shut-down sequence where logging remains active while all normal actors terminated. It monitors all user created actors.

### **3) System Guardian Actor**

This actor works same as user guardian actor except that it works for system actors. The system guardian monitors the user guardian and initiate its own shut-down upon reception of the Terminated message.

### **4) Scheduler**

Scheduler is a trait and extends to AnyRef. It is used to handle scheduled tasks. It provides the facility to schedule messages. You can schedule sending of messages and execution of tasks. It creates new instance for each ActorSystem for scheduling tasks to happen at specific time.

It returns a cancellable reference so that you can cancel the execution of the scheduled operation by calling cancel method on this reference object.

You can implement Scheduler by importing **akka.actor.Scheduler** package.

### **5) Event System**

The Event System also known as eventStream is a main event bus for each ActorSystem. It is used to carry log messages and dead latters. You can also used it to publish messages across entire ActorSystem. You can get eventStream reference by calling actorSystemRef.eventStream() method.

### **6) Configuration**

ActorSystem provides a configuration component which is used to configure application. You can access it from your actor system.

Akka Props

Props is a configuration class which is used to specify options while creating an actor. It is immutable, so it is thread-safe and shareable.

You can implement Props by importing **akka.actor.Props** package.

You can create actor by passing a Props instance into the actorOf() factory method which is available in ActorSystem and ActorContext. The actorOf() method returns an instance of ActorRef. This instance is immutable and has one to one relationship with the actor it represents. ActorRef is also serializable so that you can serialize it.

In the following example, we are creating an Actor by using Props.

Akka Actor Example: by using Props

1. **import** akka.actor.Actor;
2. **import** akka.actor.ActorSystem;
3. **import** akka.actor.Props;
4. **class** PropsExample **extends** Actor {
5. **def** receive= {
6. **case** msg:String => println(msg+" "+self.path.name)
7. }
8. }
9. **object** PropsMain{
10. **def** main(args:Array[String]){
11. **var** actorSystem = ActorSystem("ActorSystem");
12. **var** actor = actorSystem.actorOf(Props[PropsExample],"PropExample");
13. actor ! "Hello from"
14. }
15. }

**Output:**

Hello from PropExample

There are various other ways to implement Props. Let's see an example.

Akka Actor Example2: by using Props

1. **import** akka.actor.\_
2. **class** CreatingActor **extends** Actor{
3. **def** receive = {
4. **case** msg:String => println(msg+" "+self.path.name)    // Receiving message and name of actor
5. }
6. }
8. **object** CreatingActorExample{
9. **def** main(args:Array[String]){
10. **var** actorSystem = ActorSystem("ActorSystem");
11. **var** props1 = Props[CreatingActor];  // creating pops here
12. **var** actor1 = actorSystem.actorOf(props1);  // passing pops reference
13. **var** actor2 = actorSystem.actorOf(Props[CreatingActor],"Actor2")  // Passing pops and explicitly giving name to the actor
14. **var** actor3 = actorSystem.actorOf(Props(classOf[CreatingActor]),"Actor3")  // Passing actor class by using classOf
15. **var** actor4 = actorSystem.actorOf(Props[CreatingActor], name = "Actor4")  // Name passing to variable
16. **var** actor5 = actorSystem.actorOf(Props(**new** CreatingActor()), name = "Actor5") // This approach is not recommended
17. actor1 ! "Hello"
18. actor2 ! "Hello"
19. actor3 ! "Hello"
20. actor4 ! "Hello"
21. actor5 ! "Hello"
22. }
23. }

Where name is not given explicitly, it will be automatically generated.

Output:

Hello $a // Reference of the name of actor

Hello Actor2

Hello Actor4

Hello Actor3

Hello Actor5

Akka Creating Child Actor

Akka provides facility to create child actor. You can create child actor by using implicit context reference. ActorSystem is used to create root-level or top-level actor. Akka provides you context so that you can create child actor also. In the following example, we have created a child actor by using context reference.

1. **import** akka.actor.\_
2. **class** RootActor **extends** Actor{
3. **def** receive = {
4. **case** msg:String => println(msg+" "+self.path.name);
5. **var** childActor =  context.actorOf(Props[Child],"Child");
6. childActor ! "Hello"
7. }
8. }
9. **class** Child **extends** Actor{
10. **def** receive = {
11. **case** msg:String => println(msg+" "+self.path.name);
12. }
13. }
14. **object** ChildActorExample{
15. **def** main(args:Array[String]){
16. **var** actorSystem = ActorSystem();
17. **var** actor = actorSystem.actorOf(Props[RootActor],"RootActor");
18. actor ! "Hello"
19. }
20. }

# Output: Akka Actor life cycle

Akka provides life cycle methods for Actor. There are following methods to which you can override and provide specific implementation accordingly.

## Akka Actor life cycle methods

1. preStart()
2. postStop()
3. preRestart(reason: Throwable, message: Option[Any])
4. postRestart(reason: Throwable)

Akka Actor Life Cycle

### **1) preStart()**

This is overridable method, so you can override preStart() method to provide specific implementation for an Actor. It is invoked right after the starting of Actor and when an Actor is first created. In case of restart, it is called by postRestart() method.

## Akka Actor preStart() Method Example

1. **import** akka.actor.\_
2. **import** akka.actor.TypedActor.PreStart
4. **class** LifeCyleMethodsExample **extends** Actor{
5. **def** receive = {
6. **case** msg:String => println(msg+" "+self.path.name);  // Getting name of Actor
7. }
8. override **def** preStart(){    // overriding preStart method
9. println("preStart method is called");
10. }
11. }
13. **object** ActorMain{
14. **def** main(args:Array[String]){
15. **val** actorSystem = ActorSystem("ActorSystem");
16. **val** actor = actorSystem.actorOf(Props[LifeCyleMethodsExample],"RootActor");
17. actor ! "Hello"
18. }
19. }

Output:

preStart method is called

Hello RootActor

### **2) postStop()**

You can override this method also. After stopping an actor, postStop() method is called. It is an asynchronous method. This method is used to release resources after stopping the Actor. It may be used for deregistering this Actor. Messages sent to a stopped actor will be redirected to the deadLetters of the ActorSystem.

## Akka Actor PostStop() Method Example

1. **import** akka.actor.\_
2. **import** akka.actor.TypedActor.PreStart
4. **class** LifeCyleMethodsExample **extends** Actor{
5. **def** receive = {
6. **case** msg:String => println(msg+" "+self.path.name);  // Getting name of Actor
7. }
8. override **def** postStop(){    // Overriding postStop method
9. println("postStop method is called");
10. }
11. }
13. **object** ActorMain{
14. **def** main(args:Array[String]){
15. **val** actorSystem = ActorSystem("ActorSystem");
16. **val** actor = actorSystem.actorOf(Props[LifeCyleMethodsExample],"RootActor");
17. actor ! "Hello"
19. println("stopping Actor");
20. actorSystem.stop(actor);      // Stopping Actor by passing actor reference.
21. }
22. }

Output:

Hello RootActor

stopping Actor

postStop method is called

### **3) preRestart()**

Actor may be restarted in case an exception is thrown. When an actor is restarted, preRestart() method is invoked. The preRestart() method is called with the exception that caused the restart. By default it disposes of all children Actors and then calls postStop() method.

## Akka Actor preRestart() Method Example

1. println(msg+" "+self.path.name);  // Getting name of Actor
2. **var** a:**Int** =  10/0;      // ArithmethicException occurred
3. }
4. override **def** preRestart(reason:Throwable, message: **Option**[Any]){    // Overriding preRestart method
5. println("preRestart method is called");
6. println("Reason: "+reason)
7. }
8. }
10. **object** ActorMain{
11. **def** main(args:Array[String]){
12. **val** actorSystem = ActorSystem("ActorSystem");
13. **val** actor = actorSystem.actorOf(Props[LifeCyleMethodsExample],"RootActor");
14. actor ! "Hello"
15. }
16. }

Output:

Hello RootActor

preRestart method is called

Reason: java.lang.ArithmeticException: / by zero

### **4) postRestart()**

This method is invoked right after restarting of newly created Actor. It is used to allow reinitialization after an Actor crash due to exception.

## Akka Actor postRestart() Method Example

1. **import** akka.actor.\_
3. **class** LifeCyleMethodsExample **extends** Actor{
4. **def** receive = {
5. **case** msg:String => println(msg+" "+self.path.name);  // Getting name of Actor
6. **var** a:**Int** =  10/0;      // ArithmethicException occurred
7. }
8. override **def** postRestart(reason:Throwable){    // Overriding preRestart method
9. println("postRestart method is called");
10. println("Reason: "+reason)
11. }
12. }
14. **object** ActorMain{
15. **def** main(args:Array[String]){
16. **val** actorSystem = ActorSystem("ActorSystem");
17. **val** actor = actorSystem.actorOf(Props[LifeCyleMethodsExample],"RootActor");
18. actor ! "Hello"
19. }
20. }

**Output**

Hello RootActor

postRestart method is called

Reason: java.lang.ArithmeticException: / by zero

## Akka Complete LifeCyle Example

1. **import** akka.actor.{Actor,ActorSystem, Props}
3. **class** RootActor **extends** Actor{
4. **def** receive = {
5. **case** msg => println("Message received: "+msg);
6. 10/0;
7. }
8. override **def** preStart(){
9. super.preStart();
10. println("preStart method is called");
11. }
12. override **def** postStop(){
13. super.postStop();
14. println("postStop method is called");
15. }
16. override **def** preRestart(reason:Throwable, message: **Option**[Any]){
17. super.preRestart(reason, message);
18. println("preRestart method is called");
19. println("Reason: "+reason);
20. }
21. override **def** postRestart(reason:Throwable){
22. super.postRestart(reason);
23. println("postRestart is called");
24. println("Reason: "+reason);
25. }
26. }

Output:

preStart method is called

Message received: Hello

Stoping actor

postStop method is called

preRestart method is called

Reason: java.lang.ArithmeticException: / by zero

preStart method is called

postRestart is called

Reason: java.lang.ArithmeticException: / by zero

[ERROR] [01/12/2017 15:22:18.475] [ActorSystem-akka.actor.default-dispatcher-5]

[akka://ActorSystem/user/RootActor] / by zero

java.lang.ArithmeticException: / by zero

at RootActor$$anonfun$receive$1.applyOrElse(ActorLifeCyle.scala:6)

at akka.actor.Actor$class.aroundReceive(Actor.scala:496)

at RootActor.aroundReceive(ActorLifeCyle.scala:3)

at akka.actor.ActorCell.receiveMessage(ActorCell.scala:526)

at akka.actor.ActorCell.invoke(ActorCell.scala:495)

at akka.dispatch.Mailbox.processMailbox(Mailbox.scala:257)

at akka.dispatch.Mailbox.run(Mailbox.scala:224)

at akka.dispatch.Mailbox.exec(Mailbox.scala:234)

at scala.concurrent.forkjoin.ForkJoinTask.doExec(ForkJoinTask.java:260)

at scala.concurrent.forkjoin.ForkJoinPool$WorkQueue.runTask(ForkJoinPool.java:1339)

at scala.concurrent.forkjoin.ForkJoinPool.runWorker(ForkJoinPool.java:1979)

at scala.concurrent.forkjoin.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:107)

postStop method is called

Hello RootActor

Hello Child

# Akka Actor Communication

In Akka, actors communicate to each other by sending and receiving messages.

## Akka Actor Send Messages

Akka provides two predefined methods tell() and ask() for message exchange. An actor can send messages to another Actor through these following methods.

### **1) Akka Actor tell() Method**

It is used to send a message asynchronously. It does not wait and block thread for a message. It works on "fire-forget" approach. You can also use ! (bang) exclamation mark to send message. This is the preferred way of sending messages. It gives the best concurrency and scalability characteristics.

If this method is invoked from within an Actor, the sending actor reference will be implicitly passed along with the message.

If this method is invoked from an instance that is not an Actor, the sender will be deadLetters actor reference by default.

## Actor tell() Method Example

1. **import** akka.actor.{Actor,ActorSystem, Props};
3. **class** ActorExample **extends** Actor{
4. **def** receive = {
5. **case** message:String => println("Message received: "+message+ " from - "+ self.path.name);
6. println("sender:"+ sender());  // returns ActorRef
7. }
8. }
10. **object** ActorExample{
11. **def** main(args:Array[String]){
12. **val** actorSystem = ActorSystem("ActorSystem");
13. **val** actor = actorSystem.actorOf(Props[ActorExample], "RootActor");
14. actor ! "Hello"             // Sending message by using !
15. actor.tell("Hello",**null**);  // Sending message by using tell() method
16. // Sender is not passed here.
17. }
18. }

Output:

Message received: Hello from - RootActor

sender:Actor[akka://ActorSystem/deadLetters] // ActorRef refers to deadLetters

Message received: Hello from - RootActor

sender:Actor[akka://ActorSystem/deadLetters] // ActorRef refers to deadLetters

## Akka Actor tell() Method Example2

1. **import** akka.actor.{Actor,ActorSystem, Props};
3. **class** ActorExample **extends** Actor{
4. **def** receive = {
5. **case** message:String => println("Message received: "+message+ " from - "+ self.path.name);
6. println("Sender: "+sender())
7. **var** child = context.actorOf(Props[Actor2], "ChildActor");
8. child ! "Hello"

11. }
12. }
14. **class** Actor2 **extends** Actor{
15. **def** receive = {
16. **case** message:String => println("Message received: "+message+ " from - "+ self.path.name);
17. println("Sender: "+sender());
18. }
19. }
21. **object** ActorExample{
22. **def** main(args:Array[String]){
23. **val** actorSystem = ActorSystem("ActorSystem");
24. **val** actor = actorSystem.actorOf(Props[ActorExample], "RootActor");
25. actor ! "Hello"
26. }
27. }

Output:

Message received: Hello from - RootActor

Sender: Actor[akka://ActorSystem/deadLetters] // Called from outside Actor

Message received: Hello from - ChildActor

Sender: Actor[akka://ActorSystem/user/RootActor#1451914889] // Called from within Actor

## 2) Akka Actor ask Method

In akka, ask is a pattern and involves Actors as well as Futures. Ask is used to sends a message asynchronously and it returns a Future which represents a possible reply. If the actor does not reply and complete the future, it will expire after the timeout period. After timeout period, it throws an TimeoutException. You can use either ? (question mark) or ask() to send message.

You should always prefer tell method for performance and ask method, if you want response.

## Akka Actor ask Method Example

1. **import** akka.actor.{Actor,ActorSystem, Props};
2. **import** akka.util.Timeout;
3. **import** scala.concurrent.Await
4. **import** akka.pattern.ask
5. **import** scala.concurrent.duration.\_
7. **class** ActorExample **extends** Actor{
8. **def** receive = {
9. **case** message:String => println("Message recieved: "+message);
10. }
11. }
13. **object** ActorExample{
14. **def** main(args:Array[String]){
15. **val** actorSystem = ActorSystem("ActorSystem");
16. **val** actor = actorSystem.actorOf(Props[ActorExample], "RootActor");
17. implicit **val** timeout = Timeout(2 seconds);
18. **val** future = actor ? "Hello";
19. **val** result = Await.result(future, timeout.duration);
20. println(result);
21. }
22. }

Output:

Message recieved: Hello

Exception in thread "main" java.util.concurrent.TimeoutException: Futures timed out after [2 seconds]

## Akka Actor ask() Method Example2

1. **import** akka.actor.{Actor,ActorSystem, Props, ActorRef};
2. **import** akka.util.Timeout;
3. **import** scala.concurrent.Await
4. **import** akka.pattern.ask
5. **import** scala.concurrent.duration.\_
7. **class** ActorExample **extends** Actor{
8. **def** receive = {
9. **case** message:String => println("Message received: "+message+" from outside actor instance");
10. println("Replaying");
11. **val** senderName = sender();
12. senderName ! "Hello, I got your message.";      // Replying message
13. }
14. }
16. **object** ActorExample{
17. **def** main(args:Array[String]){
18. **val** actorSystem = ActorSystem("ActorSystem");
19. **val** actor = actorSystem.actorOf(Props[ActorExample], "RootActor");
20. implicit **val** timeout = Timeout(10 seconds);
21. **val** future = actor ? "Hello";
22. **val** result = Await.result(future, timeout.duration);
23. println("Message received: "+result);
24. }
25. }

Output:

Message received: Hello from outside actor instance

Replaying

Message received: Hello, I got your message.

## Akka Actor Ask Method Example3

1. **import** akka.actor.{Actor,ActorSystem, Props};
2. **import** akka.util.Timeout;
3. **import** scala.concurrent.Await
4. **import** akka.pattern.ask
5. **import** scala.concurrent.duration.\_
7. **class** ActorExample **extends** Actor{
8. **def** receive = {
9. **case** message:String => println("Message received: "+message+" from outside actor instance");
10. Thread.sleep(5000);      // actor thread is sleeping
11. println("Replaying");
12. **val** senderName = sender();
13. senderName ! "Hello, I got your message.";      // Replying message
14. }
15. }
17. **object** ActorExample{
18. **def** main(args:Array[String]){
19. **val** actorSystem = ActorSystem("ActorSystem");
20. **val** actor = actorSystem.actorOf(Props[ActorExample], "RootActor");
21. implicit **val** timeout = Timeout(2 seconds);
22. **val** future = actor ? "Hello";
23. **val** result = Await.result(future, timeout.duration);
24. println("Message received: "+result);
25. }
26. }

Output:

Message received: Hello from outside actor instance

Exception in thread "main" java.util.concurrent.TimeoutException: Futures timed out after [2 seconds]

Akka Actor Reply Messages

You can reply to a message by using sender() method. This method gives you ActorRef which is a instance of sender Actor. You can also store this reference. Replying to a message is simple, you just need to use sender() ! ReplyMsg.

If there is no sender i.e. a message which is sent without an actor or future context, the default sender is a 'dead-letter' actor reference.

Akka Actor Replying Message Example

1. **import** akka.actor.{Actor,ActorSystem, Props};
2. **class** ActorReplyExample **extends** Actor{
3. **def** receive = {
4. **case** message:String => println("Message recieved from "+sender.path.name+" massage: "+message);
5. **val** child = context.actorOf(Props[ActorChildReplyExample],"ActorChild");
6. child ! "Hello Child"
7. }
8. }

11. **class** ActorChildReplyExample **extends** Actor{
12. **def** receive ={
13. **case** message:String => println("Message recieved from "+sender.path.name+" massage: "+message);
14. println("Replying to "+sender().path.name);
15. sender()! "I got you message";
16. }
17. }
19. **object** ActorReplyExample{
20. **def** main(args:Array[String]){
21. **val** actorSystem = ActorSystem("ActorSystem");
22. **val** actor = actorSystem.actorOf(Props[ActorReplyExample], "RootActor");
23. actor ! "Hello";
24. }
25. }

Output:

Message recieved from deadLetters massage: Hello

Message recieved from RootActor massage: Hello Child

Replying to RootActor

Message recieved from ActorChild massage: I got you message

Akka Actor Forward Message

You can forward a message from one actor to another. In this case, address/reference of an Actor is maintained even though the message is going through a 'mediator'.

It is helpful when writing actors that work as routers, load-balancers, replicators etc.

Akka Actor Forward Message Example

1. **import** akka.actor.{Actor,ActorSystem, Props};
3. **class** ActorExample **extends** Actor{
4. **def** receive = {
5. **case** message:String => println("Message received from "+sender().path.name+" : "+message);
6. **val** child = context.actorOf(Props[Actor2],"ChildActor");
7. println("message forwarded to child Actor");
8. child ! message ;    // Message forwarded to child actor
10. **case** \_ => println("Unknown message");
11. }
12. }
14. **class** Actor2 **extends** Actor{
15. **def** receive ={
16. **case** message:String => println("Message received from "+sender().path.name+" : "+message);
17. **case** \_ => println("Unknown message");
18. }
19. }

22. **object** ActorExample{
23. **def** main(args:Array[String]){
24. **val** actorSystem = ActorSystem("ActorSystem");
25. **val** actor = actorSystem.actorOf(Props[ActorExample], "RootActor");
26. actor ! "Hello"
27. }
28. }

Output:

Message received from deadLetters : Hello

message forwarded to child Actor

Message received from RootActor : Hello

Akka Stopping Actors

In Akka, you can stop Actors by invoking the stop() method of either ActorContext or ActorSystem class. ActorContext is used to stop child actor and ActorSystem is used to stop top level Actor.

The actual termination of the actor is performed asynchronously.

There are some other methods available in Akka, which are used to stop Actor. Some of which are PoisonPill, terminate() and gracefulStop() are used to stop Actor.

Akka Stopping Top Level Actor Example

You can simply call stop() method to stop top level Actor. The following example describes the uses of stop() method.

1. **import** akka.actor.{Actor,ActorSystem, Props};
3. **class** ActorExample **extends** Actor{
4. **def** receive = {
5. **case** message:String => println("Message received: "+message);
6. **case** \_ => println("Unknown message");
7. }
8. override **def** postStop(){
9. println("Actor stoped");
10. }
11. }
13. **object** ActorExample{
14. **def** main(args:Array[String]){
15. **val** actorSystem = ActorSystem("ActorSystem");
16. **val** actor = actorSystem.actorOf(Props[ActorExample], "RootActor");
17. actor ! "Hello"
18. actorSystem.stop(actor);
19. }
20. }

Output:

Message received: Hello

Actor stoped

Akka Stopping Child Actor Example

1. **import** akka.actor.{Actor,ActorSystem, Props};
3. **class** ActorExample **extends** Actor{
4. **def** receive = {
5. **case** message:String => println("Message received by "+self.path.name+": "+message);
6. **val** childactor = context.actorOf(Props[ChildActor], "ChildActor");
7. childactor ! "Hello child Actor"
8. context.stop(childactor);
10. **case** \_ => println("Unknown message");
11. }
12. }

15. **class** ChildActor **extends** Actor{
16. **def** receive = {
17. **case** message:String => println("Message received by "+self.path.name+": "+message);
18. **case** \_ => println("Unknown message");
19. }
21. override **def** postStop(){
22. println("Child Actor stoped");
23. }
24. }

27. **object** ActorExample{
28. **def** main(args:Array[String]){
29. **val** actorSystem = ActorSystem("ActorSystem");
30. **val** actor = actorSystem.actorOf(Props[ActorExample], "RootActor");
31. actor ! "Hello"
33. }
34. }

Output:

Message received by RootActor: Hello

Message received by ChildActor: Hello child Actor

Child Actor stoped

Akka Stopping ActorSystem Example

You can stop actor system by calling it's terminate method. This method will stop the guardian actor, which in turn will recursively stop all its child actors.

1. **import** akka.actor.{Actor,ActorSystem, Props};
3. **class** ActorExample **extends** Actor{
4. **def** receive = {
5. **case** message:String => println("Message received by "+self.path.name+": "+message);
6. **val** childactor = context.actorOf(Props[ChildActor], "ChildActor");
7. childactor ! "Hello child Actor"
9. **case** \_ => println("Unknown message");
10. }
12. override **def** postStop(){
13. println("Top Level Actor stoped");
14. }
15. }

18. **class** ChildActor **extends** Actor{
19. **def** receive = {
20. **case** message:String => println("Message received by "+self.path.name+": "+message);
21. **case** \_ => println("Unknown message");
22. }
24. override **def** postStop(){
25. println("Child Actor stoped");
26. }
27. }

30. **object** ActorExample{
31. **def** main(args:Array[String]){
32. **val** actorSystem = ActorSystem("ActorSystem");
33. **val** actor = actorSystem.actorOf(Props[ActorExample], "RootActor");
34. actor ! "Hello"
35. actorSystem.terminate();
36. }
37. }

Output:

Message received by RootActor: Hello

Message received by ChildActor: Hello child Actor

Child Actor stoped

Top Level Actor stoped

# Scala Interview Questions

### **1) What is Scala?**

Scala is a general-purpose programming language. It supports object oriented, functional and imperative programming approaches. It is a strong static type language. In Scala, everything is an object whether it is a function or a number.It was designed by Martin Odersky in 2004.

For more information: [Click here](http://www.javatpoint.com/what-is-scala)

### **2) What are the features of Scala?**

There are following features in Scala:

* Type inference
* Singleton object
* Immutability
* Lazy computation
* Case classes and Pattern matching
* Concurrency control
* String interpolation
* Higher order function
* Traits
* Rich collection set and many more.

For more information: [Click here](http://www.javatpoint.com/features-of-scala)

### **3) What are the 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 in the tutorials.

For more information: [Click here](http://www.javatpoint.com/scala-variable-and-data-types)

### **4) What is pattern matching?**

Pattern matching is a feature of Scala. It works same as switch case in other languages. It matches best case available in the pattern.

For more information: [Click here](http://www.javatpoint.com/scala-pattern-matching)

### **5) What is for-comprehension in Scala?**

In Scala, for loop is known as for-comprehensions. It can be used to iterate, filter and return an iterated collection. The for-comprehension looks a bit like a for-loop in imperative languages, except that it constructs a list of the results of all iterations.

For more information: [Click here](http://www.javatpoint.com/scala-for-loop)

### **6) What is breakable method in Scala?**

In Scala, there is no break statement but you can do it by using break method and importing Scala.util.control.Breaks.\_ package. It can break your code.

For more information: [Click here](http://www.javatpoint.com/scala-break)

### **7) How to declare function in Scala?**

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.

For more information: [Click here](http://www.javatpoint.com/scala-functions)

### **8) Why do we use =(equal) operator in Scala function.**

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.

For more information: [Click here](http://www.javatpoint.com/scala-functions)

### **9) What is Function parameter with default value in Scala?**

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.

For more information: [Click here](http://www.javatpoint.com/scala-functions)

### **10) What is function named parameter in Scala?**

In Scala function, you can specify the names of parameters during calling the function. You can pass named parameters in any order and can also pass values only.

For more information: [Click here](http://www.javatpoint.com/scala-functions)

### **11) What is higher order function in Scala?**

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.

For more information: [Click here](http://www.javatpoint.com/scala-higher-order-functions)

### **12) What is function composition in Scala?**

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.

For more information: [Click here](http://www.javatpoint.com/scala-higher-order-functions)

### **13) What is Anonymous (lambda) Function in Scala?**

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.

For more information: [Click here](http://www.javatpoint.com/scala-higher-order-functions)

### **14) What is multiline expression in Scala?**

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

For more information: [Click here](http://www.javatpoint.com/scala-higher-order-functions)

### **15) What is function 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.

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

For more information: [Click here](http://www.javatpoint.com/scala-higher-order-functions)

### **16) What is nexted function in Scala?**

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.

For more information: [Click here](http://www.javatpoint.com/scala-higher-order-functions)

### **17) What is object in Scala?**

Object is a real world entity. It contains state and behavior. Laptop, car, cell phone are the real world objects. Object typically has two characteristics:

**1) State:** data values of an object are known as its state.

**2) Behavior:** functionality that an object performs is known as its behavior.

Object in Scala is an instance of class. It is also known as runtime entity.

For more information: [Click here](http://www.javatpoint.com/scala-object-and-class)

### **18) What is class in Scala?**

Class is a template or a blueprint. It is also known as collection of objects of similar type.

In Scala, a class can contain:

1. Data member
2. Member method
3. Constructor
4. Block
5. Nested class
6. Super class information etc.

For more information: [Click here](http://www.javatpoint.com/scala-object-and-class)

### **19) What is anonymous object in Scala?**

In Scala, you can create anonymous object. An object which has no reference name is called anonymous object. It is good to create anonymous object when you don't want to reuse it further.

For more information: [Click here](http://www.javatpoint.com/scala-object-and-class)

### **20) What is constructor in Scala?**

In Scala, constructor is not special method. Scala provides primary and any number of auxiliary constructors.

In Scala, if you don't specify primary constructor, compiler creates a default primary constructor. All the statements of class body treated as part of constructor. It is also known as default constructor.

For more information: [Click here](http://www.javatpoint.com/scala-constructor)

### **21) What is method overloading in Scala?**

Scala provides method overloading feature which allows us to define methods of same name but having different parameters or data types. It helps to optimize code. You can achieve method overloading either by using different parameter list or different types of parameters.

For more information: [Click here](http://www.javatpoint.com/scala-method-overloading)

### **22) What is this in Scala?**

In Scala, this is a keyword and used to refer current object. You can call instance variables, methods, constructors by using this keyword.

For more information: [Click here](http://www.javatpoint.com/scala-this)

### **23) What is Inheritance?**

Inheritance is an object oriented concept which is used to reusability of code. You can achieve inheritance by using extends keyword. To achieve inheritance a class must extend to other class. A class which is extended called super or parent class. a class which extends class is called derived or base class.

For more information: [Click here](http://www.javatpoint.com/scala-inheritance)

### **24) What is method overriding in Scala?**

When a subclass has the same name method as defined in the parent class, it is known as method overriding. When subclass wants to provide a specific implementation for the method defined in the parent class, it overrides method from parent class.

In Scala, you must use either override keyword or override annotation to override methods from parent class.

For more information: [Click here](http://www.javatpoint.com/scala-method-overriding)

### **25) What is final in Scala?**

Final keyword in Scala is used to prevent inheritance of super class members into derived class. You can declare final variable, method and class also.

For more information: [Click here](http://www.javatpoint.com/scala-final)

### **26) What is final class in Scala?**

In Scala, you can create final class by using final keyword. Final class can't be inherited. If you make a class final, it can't be extended further.

For more information: [Click here](http://www.javatpoint.com/scala-final)

### **27) What is abstract class in Scala?**

A class which is declared with abstract keyword is known as abstract class. An abstract class can have abstract methods and non-abstract methods as well. Abstract class is used to achieve abstraction.

For more information: [Click here](http://www.javatpoint.com/scala-abstract-class)

### **28) What is Scala Trait?**

A trait is like an interface with a partial implementation. In Scala, trait is a collection of abstract and non-abstract methods. You can create trait that can have all abstract methods or some abstract and some non-abstract methods.

For more information: [Click here](http://www.javatpoint.com/scala-trait)

### **29) What is trait mixins in Scala?**

In Scala, trait mixins means you can extend any number of traits with a class or abstract class. You can extend only traits or combination of traits and class or traits and abstract class.

It is necessary to maintain order of mixins otherwise compiler throws an error.

For more information: [Click here](http://www.javatpoint.com/scala-trait-mixins)

### **30) What is access modifier in Scala?**

Access modifier is used to define accessibility of data and our code to the outside world. You can apply accessibly to class, trait, data member, member method and constructor etc. Scala provides least accessibility to access to all. You can apply any access modifier to your code according to your requirement.

In Scala, there are only three types of access modifiers.

1. No modifier
2. Protected
3. Private

For more information: [Click here](http://www.javatpoint.com/scala-access-modifier)

### **31) What is array in Scala?**

In Scala, array is a combination of mutable values. It is an index based data structure. It 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. Scala arrays also support all the sequence operations.

For more information: [Click here](http://www.javatpoint.com/scala-array)

### **32) What is ofDim method in Scala?**

Scala provides an ofDim method to create multidimensional array. Multidimensional array is an array which store data in matrix form. You can create from two dimensional to three, four and many more dimensional array according to your need.

For more information: [Click here](http://www.javatpoint.com/scala-multidimensional-array)

### **33) What is String in Scala?**

In Scala, string is a combination of characters or we can say it is a sequence of characters. It is index based data structure and use linear approach to store data into memory. String is immutable in Scala like java.

For more information: [Click here](http://www.javatpoint.com/scala-string)

### **34) What is string interpolation in Scala?**

Starting in Scala 2.10.0, Scala offers a new mechanism to create strings from your data. It is called string interpolation. String interpolation allows users to embed variable references directly in processed string literals. Scala provides three string interpolation methods: s, f and raw.

For more information: [Click here](http://www.javatpoint.com/scala-string-interpolation)

### **35) What does s method in Scala String interpolation?**

The s method of string interpolation allows us to pass variable in string object. You don't need to use + operator to format your output string. This variable is evaluated by compiler and variable is replaced by value.

For more information: [Click here](http://www.javatpoint.com/scala-string-interpolation)

### **36) What does f method in Scala String interpolation?**

The f method is used to format your string output. It is like printf function of C language which is used to produce formatted output. You can pass your variables of any type in the print function.

For more information: [Click here](http://www.javatpoint.com/scala-string-interpolation)

### **37) What does raw method in Scala String interpolation?**

The raw method of string interpolation is used to produce raw string. It does not interpret special char present in the string.

For more information: [Click here](http://www.javatpoint.com/scala-string-interpolation)

### **38) What is exception handling in Scala?**

Exception handling is a mechanism which is used to handle abnormal conditions. You can also avoid termination of your program unexpectedly.

Scala makes "checked vs unchecked" very simple. It doesn't have checked exceptions. All exceptions are unchecked in Scala, even SQLException and IOException.

For more information: [Click here](http://www.javatpoint.com/scala-exception-handling)

### **39) What is try catch in Scala?**

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.

For more information: [Click here](http://www.javatpoint.com/scala-try-catch)

### **40) What is finally in Scala?**

The finally block is used to release resources during exception. Resources may be file, network connection, database connection etc. the finally block executes guaranteed.

For more information: [Click here](http://www.javatpoint.com/scala-finally-block)

### **41) What is throw in Scala?**

You can throw exception explicitly in you code. Scala provides throw keyword to throw exception. The throw keyword mainly used to throw custom exception.

For more information: [Click here](http://www.javatpoint.com/scala-throw-keyword)

### **42) What is exception propagation in Scala?**

In Scala, you can propagate exception in calling chain. When an exception occurs in any function it looks for handler. If handler not available there, it forwards to caller method and look for handler there. If handler present there, handler catch that exception. If handler not present it moves to next caller method in calling chain. This whole process is known as exception propagation.

### **43) What is throws in Scala?**

Scala provides throws keyword to declare exception. You can declare exception with method definition. It provides information to the caller function that this method may throw this exception. It helps to caller function to handle and enclose that code in try-catch block to avoid abnormal termination of program. In Scala, you can either use throws keyword or throws annotation to declare exception.

For more information: [Click here](http://www.javatpoint.com/scala-throws-keyword)

### **44) What is custom exception in Scala?**

In Scala, you can create your own exception. It is also known as custom exceptions. You must extend Exception class to while declaring custom exception class. You can create your own message in custom class.

For more information: [Click here](http://www.javatpoint.com/scala-custom-exception)

### **45) What is collection in Scala?**

Scala provides rich set of collection library. It contains classes and traits to collect data. These collections can be mutable or immutable. You can use them according to your requirement.

For more information: [Click here](http://www.javatpoint.com/scala-collection)

### **46) What is traversable in Scala collection?**

It is a trait and used to traverse collection elements. It is a base trait for all Scala collections. It contains the methods which are common to all collections.

For more information: [Click here](http://www.javatpoint.com/scala-collection)

### **47) What does Set in Scala collection?**

It is used to store unique elements in the set. It does not maintain any order for storing elements. You can apply various operations on them. It is defined in the Scala.collection.immutable package.

for more information: [Click here](http://www.javatpoint.com/scala-set)

### **48) What does SortedSet in Scala collection?**

In Scala, SortedSet extends Set trait and provides sorted set elements. It is useful when you want sorted elements in the Set collection. You can sort integer values and string as well.

It is a trait and you can apply all the methods defined in the traversable trait and Set trait.

for more information: [Click here](http://www.javatpoint.com/scala-set)

### **49) What is HashSet in Scala collection?**

HashSet is a sealed class. It extends AbstractSet and immutable Set trait. It uses hash code to store elements.

It neither maintains insertion order nor sorts the elements.

For more information: [Click here](http://www.javatpoint.com/scala-hashset)

### **50) What is BitSet in Scala?**

Bitsets are sets of non-negative integers which are represented as variable-size arrays of bits packed into 64-bit words. The memory footprint of a bitset is determined by the largest number stored in it. It extends Set trait.

For more information: [Click here](http://www.javatpoint.com/scala-bitset)

### **51) What is ListSet in Scala collection?**

In Scala, ListSet class implements immutable sets using a list-based data structure.

Elements are stored internally in reversed insertion order, which means the newest element is at the head of the list. It maintains insertion order.

This collection is suitable only for a small number of elements.

For more information: [Click here](http://www.javatpoint.com/scala-listset)

### **52) What is Seq in Scala collection?**

Seq is a trait which represents indexed sequences that are guaranteed immutable. You can access elements by using their indexes. It maintains insertion order of elements.

Sequences support a number of methods to find occurrences of elements or subsequences.

It returns a list.

For more information: [Click here](http://www.javatpoint.com/scala-seq)

### **53) What is Vector in Scala collection?**

Vector is a general-purpose, immutable data structure. It provides random access of elements. It is good for large collection of elements.

It extends an abstract class AbstractSeq and IndexedSeq trait.

For more information: [Click here](http://www.javatpoint.com/scala-vector)

### **54) What is List in Scala Collection?**

List is used to store ordered elements. It extends LinearSeq trait. It is a class for immutable linked lists. This class is good for last-in-first-out (LIFO), stack-like access patterns.

It maintains order, can contain duplicates elements.

For more information: [Click here](http://www.javatpoint.com/scala-list)

### **55) What is Queue in Scala Collection?**

Queue implements a data structure that allows inserting and retrieving elements in a first-in-first-out (FIFO) manner.

In Scala, Queue is implemented as a pair of lists. One is used to insert the elements and second to contain deleted elements. Elements are added to the first list and removed from the second list.

For more information: [Click here](http://www.javatpoint.com/scala-queue)

### **56) What is stream in Scala?**

Stream is a lazy list. It evaluates elements only when they are required. This is a feature of Scala. Scala supports lazy computation. It increases performance of your program.

For more information: [Click here](http://www.javatpoint.com/scala-stream)

### **57) What does Map in Scala Collection?**

Map is used to store elements. It stores elements in pairs of key and values. In Scala, you can create map by using two ways either by using comma separated pairs or by using rocket operator.

For more information: [Click here](http://www.javatpoint.com/scala-maps)

### **58) What does ListMap in Scala?**

This class implements immutable maps by using a list-based data structure. It maintains insertion order and returns ListMap. This collection is suitable for small elements.

You can create empty ListMap either by calling its constructor or using ListMap.empty method.

For more information: [Click here](http://www.javatpoint.com/scala-listmap)

### **59) What is tuple in Scala?**

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 to mix type data. You can return multiple values by using tuple in function.

For more information: [Click here](http://www.javatpoint.com/scala-tuples)

### **60) What is singleton object in Scala?**

Singleton object is an object which is declared by using object keyword instead by class. No object is required to call methods declared inside singleton object.

In Scala, there is no static concept. So Scala creates a singleton object to provide entry point for your program execution.

For more information: [Click here](http://www.javatpoint.com/scala-singleton-and-companion-object)

### **61) What is companion object in Scala?**

In Scala, when you have a class with same name as singleton object, it is called companion class and the singleton object is called companion object.

The companion class and its companion object both must be defined in the same source file.

For more information: [Click here](http://www.javatpoint.com/scala-singleton-and-companion-object)

### **62) What are case classes in Scala?**

Scala case classes are just regular classes which are immutable by default and decomposable through pattern matching.

It uses equal method to compare instance structurally.

It does not use new keyword to instantiate object.

For more information: [Click here](http://www.javatpoint.com/scala-case-classes-and-case-object)

### **63) What is file handling in Scala?**

File handling is a mechanism of handling file operations. Scala provides predefined methods to deal with file. You can create, open, write and read file. Scala provides a complete package scala.io for file handling.