This Scala material prepared by **Nireekshan** with **Arjun** guidelines and reviewed by **Ramesh** sir @DVS

By Nireekshan

Feel free to contact if any queries:

Primary : dvs.training@gmail.com
Secondary : nirekshan@gmail.com

Nireekshan

- ✓ This scala document is divided into two parts,
  - **PART – 1** - Regular
    - During sessions for understanding the topics
  - **PART – 2** - Practice
    - This doc contains only programs based on chapter wise.
    - These programs need to practice on Daily
- ✓ Make sure you guys having both Part-1 and Part-2 docs to proceed for preparation

PART – 1: Regular

SCALA INDEX

- What is mutable?
- Creating variable by using var
- Few points to make a note
- When should we go for var variable?
- Conclusion for var
- Multiple variable initializations

✓ val keyword

- What is immutable?
- Creating variable by using val
- Few points to make a note
- When should we go for val variable?
- Conclusion for val

✓ Multiple variable initializations
✓ Type inference
✓ null value
✓ Summary of the story:

6. Data types in Scala

✓ What is a data type?
✓ Scala data type's image
✓ Data type
✓ What is the default package in scala?
✓ Mostly usage classes from scala package
✓ Types of data types
✓ Numeric data types

- Byte
- Short
- Int
- Long

✓ Floating Point Data types:

- Float
- Double

✓ Char Data types
✓ Boolean Data types:
✓ Summary

7. Flow control

✓ Why should we learn about flow control?
✓ Flow control
✓ Sequential
✓ Conditional
✓ Looping
✓ Sequential statements
✓ Conditional or Decision-making statements
✓ Looping
✓ Others

✓ Sequential statements

- if statement
- When should we use if statement?
- if else statement
- When should we use if statement?

- if else if statement
- When should we use if statement?

✓ Looping

- do while
- while
- for loop (for comprehension or for expression)
- Difference between until and to
- Scala for-loop example using by keyword
- Scala for-loop filtering example
- Scala for-loop example by using yield keyword
- Scala for-loop in Collection

8. String in Scala

✓ What is a String?
✓ How to create String object?
✓ String literal.
✓ new keyword.
✓ Methods in String class
✓ Important methods for String class:

- public int length()
- public String toLowerCase()
- public String toUpperCase()
- public String concat(String s)
- public boolean equals(Object obj)
- public String substring(int begin)
- public char charAt(int index)
- public boolean equalsIgnoreCase(String s)
- public String substring(int begin , int end)
- public String replace(char old, char new)
- public String trim()

✓ String interpolation

Object Oriented Programming in Scala

9. OOPS – Part – 1 class, object, method etc

✓ OOPs (Object Oriented Programming Principles)
  - class
  - object

- Why should we create an object for a class?
- What is an Object?
- Make some notes
- characteristics

✓ Data Hiding or Information Hiding:
✓ Abstraction
✓ Encapsulation:
✓ Methods
  - Types of methods

- Zero parameterized methods
- Parameterized methods

✓ Constructors in scala

- What is the purpose of constructor?
- When constructor will get execute?
- How many times constructor will get execute?
- Does developer need to call constructor explicitly like a method?
- Types of constructor
- Without parameters Primary constructor
- Primary constructor which are having parameters
- Auxiliary Constructor

10. OOPS – Part – 2 Inheritance

✓ Inheritance
- What is inheritance?
- How to implement inheritance?
- Still expecting more explanation then…
- Advantages of Inheritance:
- Types of Inheritance
  - Single Inheritance
  - Multi-level Inheritance
  - Multiple Inheritance
  - Why multiple inheritance is not supporting?

11. OOPS – Part – 3 Polymorphism

✓ Polymorphism

- What is polymorphism
- Dynamic Polymorphism
- Method Overloading
- Cases in overloading
  - Difference in the number of parameters.
  - Difference in the datatype of parameters.
  - Difference in the order or sequence of parameters.
- Can we overload main() method?
- Method overriding
- When should we go for overriding? (Please don't say as I don't know)
- Difference between Method overloading and Method overriding
- final keyword
  - final method
  - final class
  - Smart question: If we are using final keyword then are, we missing OOPs features?

12. OOPS – Part – 4 abstract class, trait

- ✓ Abstract class
  - Abstract keyword
  - Types of methods
    - Implemented method
    - Unimplemented method
  - Abstract method
  - Abstract class
  - Abstract variable
  - If you have time
    - Please prepare given scenarios
- ✓ trait
  - trait keyword
  - What is trait?
  - A single class can extends multiple traits
  - If you have time
    - Please prepare given scenarios

13. OOPS – Part – 5 Normal class, Singleton object, Companion object etc

- ✓ Normal class, Singleton object and Standalone class
  - Normal class
  - Singleton object
  - Standalone class
- ✓ Singleton object
  - Purpose of singleton object
  - Difference between instance variable and singleton variable
  - How to access singleton variable
- ✓ Companion object
  - What is companion object
  - Advantage
  - Rules to define companion object
- ✓ Case class
  - Case keyword
  - Why case class?
  - Advantage
  - Difference between case class and normal class

14. Functional Programming

- ✓ General example why function required
- ✓ When should we go for function?
- ✓ What is a Function?
- ✓ Advantages
- ✓ User defined functions
- ✓ Function related terminology
- ✓ Main parts in Function
    - Defining or creating a function
    - Calling a function
- ✓ Syntax surprise - 1
- ✓ Functions are two types
    - Function without parameters
    - Function with parameters
- ✓ A Function can call other function
- ✓ return keyword in scala
    - Function without return statement
    - Function with return statement
    - If function having return statement then,
    - Important point on return statement
    - Why we need to assign function calling to a variable?
    - return vs Unit type
    - Unit type
    - Syntax surprise - 2
    - Important point about return statement
- ✓ Function Parameters with default Values
- ✓ Scala Function Named Parameter
- ✓ Scala functions are first class values
- ✓ Higher Order Functions
- ✓ What is higher order function?
- ✓ Usage of higher order functions
- ✓ Case 1: Passing a Function as Parameter in a Function
- ✓ Case 2: A function can return another function
- ✓ Case 3: Function composing
- ✓ Case 4: Scala Anonymous (lambda) Function
- ✓ Purpose of anonymous or lambda functions?
- ✓ Creating anonymous function
- ✓ Anonymous function by using _ (underscore) wild card
- ✓ Scala Function Currying
- ✓ Function with Variable Length Parameters
- ✓ Nested Functions

15. Collection Framework

- ✓ Why should we go for collection framework?
- ✓ Collection library
- ✓ Scala List
- ✓ Adding two List objects
- ✓ List is immutable
- ✓ ListBuffer
- ✓ Iterating elements in List
- ✓ Scala Set
- ✓ Set information
- ✓ Adding two Set objects
- ✓ Searching for specific object in Set
- ✓ Adding and Removing elements in Set

- ✓ Iterating elements in Set
- ✓ SortedSet maintains order
- ✓ Scala Maps
- ✓ Ways to create a Map
- ✓ Adding and Removing elements to Map
- ✓ Make a note:

16. Tuples

- ✓ Scala Tuples
- ✓ Accessing values from tuple

17. Exception Handling

- ✓ Normal flow of the execution
- ✓ Abnormal flow of the execution
- ✓ What we need to do if program terminates abnormally?
- ✓ What is an Exception?
- ✓ Is it really required to handle the exceptions?
- ✓ What is the meaning of exception handling?
- ✓ Handling exceptions by using try catch
- ✓ try with multiple case blocks inside catch block
- ✓ finally block
- ✓ What is the speciality of final block?
- ✓ throw keyword or Creating customized exceptions
- ✓ Rules to create customized exception

1. Scala Programming Introduction

1. What is Scala?

- ✓ Scala is a general purpose and high-level programming language.
    - General purpose means, all companies are using Scala programming language to develop the applications, testing and maintenance etc.
    - There are mainly two types of programming languages,
        - High level
            - Human readable language.
            - Easy to understand
        - Low level
            - Machine readable language like bits (1's and 0's form)

2. Where are all Scala is using in application level?

To develop,

- ✓ Standalone applications
    - An application which needs to install on every machine to work with that application.
- ✓ Web applications
    - An application which follows client-server architecture.
    - Client is a program, which sends request to the server.
    - Server is a program, mainly it can do three things,
        - Captures the request from client
        - Process the request
        - Sends the response to the client
- ✓ Database applications.
- ✓ To process huge amount of data.
    - Hadoop
    - Spark.
- ✓ Machine learning.
- ✓ Artificial Intelligence.
- ✓ Data science.

3. Before Scala, Functional programming was existing but,

- ✓ Functional programming language is the process of building software by using,
  - Functions
  - Immutability
  - Composing functions
  - Higher order functions
  - Pattern matching etc.

- ✓ Limitation: Functional programming language is missing the Object-Oriented Programming principles.

4. Before Scala, Object-Oriented Programming also existing but,

- ✓ Object oriented programming language is the process of building software by using,
  - Classes
  - Objects
  - Inheritance
  - Polymorphism
  - Data hiding
  - Abstraction etc.

- ✓ Limitation: Object oriented programming language is missing the Functional Programming language features.

5. After Scala simple definition for FP and OOPs

- ✓ Scala = Functional programming + Object Oriented programming.
- ✓ Scala was designed to be both object-oriented and functional.

- ✓ It is a pure object-oriented language means every value is an object.
  - Objects are defined by classes.

- ✓ Scala is also a functional language means,
  - Every function is a value.
  - Functions can be nested
  - They can operate on data using pattern matching.

- ✓ Scala programs run on top of Java Virtual Machine (JVM).
- ✓ JVM is a program which converts byte code (.class) instructions into machine understandable format.

6. History of Scala?

- ✓ Scala was created by Martin Odersky.
- ✓ Martin Odersky was,
  - Co-designer of Java generics.
  - The original author of the current javac reference compiler.
- ✓ Initially first release was in the year of 2004.

7. Scala supports

- ✓ Functional programming.
- ✓ Object oriented programming approach
- ✓ Now days to fulfil the requirement both are required.

Scala = Functional programming + Object oriented programming

8. Which companies are using scala?

- ✓ Currently all companies are using the Scala.

9. Machine language

- ✓ Representing the instructions and data in the form of *bits* (1's and 0's) is called machine code or machine language.
- ✓ Example to add two numbers then machine will convert these numbers into bits by division with 2.

Ex : 12 + 14 = 26

2	|	12		
	|	6	- 0	Reminder
	|	3	- 0	Reminder
	|	1	- 1	Reminder

✓ 12 == 1100 Take the digits from bottom to top digits

2	|	14		
	|	7	- 0	Reminder
	|	3	- 1	Reminder
	|	1	- 1	Reminder

✓ 14 == 1110 Take the digits from bottom to top digits

- ✓ Internally these bit values will be adding and generate the sum result as 26

9.1 Translator

- ✓ A Translator is a program that converts any computer programs into machine code.
- ✓ There are 'n' number of translators are existing but for us we need to understand 2 types of translators.

9.1.1. Interpreter

- ✓ Interpreter is a program; it can convert the program line by line.

9.1.2. Compiler

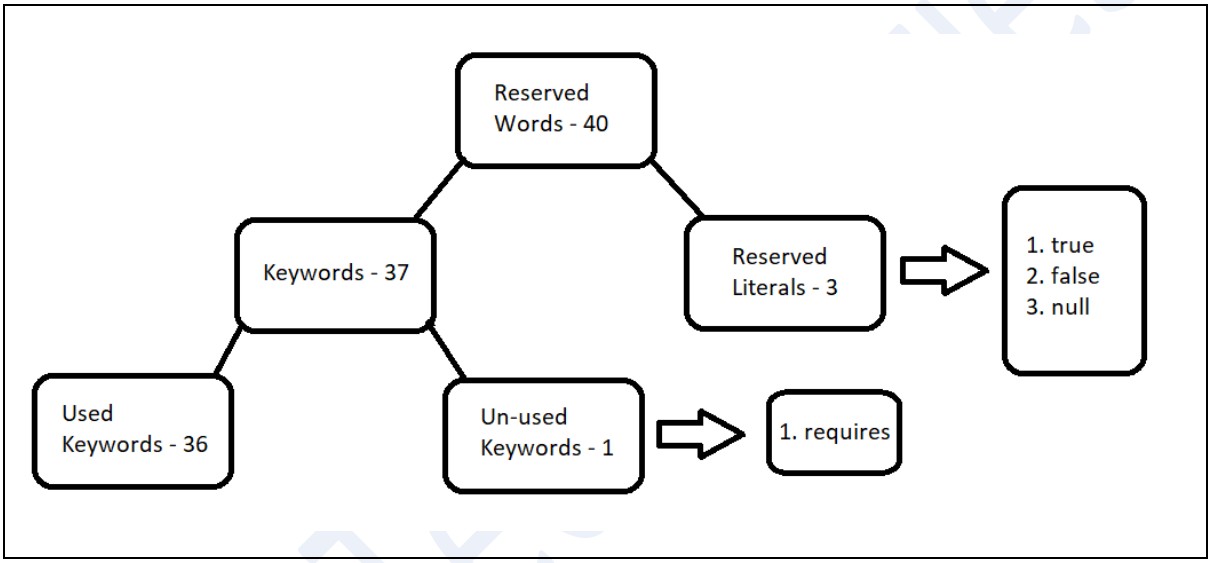
- ✓ Compiler is a program converts the entire program in a single step.

2. Scala keywords and important components in program

1. I'm sure 99.9999% a scala program contains below things,

1.1 What is reserved word or keyword?

- ✓ The words which are reserved to do a specific functionality is called as reserved words also called as keywords.
- ✓ There are total 40 reserved words in scala programming language
  - These are divided into two types
    - Keywords - 37
    - Reserved literals - 3



Make a note

- ✓ For your better understanding i have created a table and please follow that.

1.2 Scala keywords table

Flow Control	Access Modifiers	Exception Handling	class related	object related	Function related	Variable related	Un-used related	Reserved literal
if	private	try	import	new	def	val	requires	true
else	protected	catch	package	this		var		false
do	abstract	finally	class	super				null
while	final	throw	extends					
for	lazy		type					
yield	sealed		trait					
match	implicit		object					
case	override		with					
return			forSome					
9	**8**	**4**	**9**	**3**	**1**	**2**	**1**	**3**

Keywords count down

- ✓ 9 + 8 + 4 + 9 + 3 + 1 + 2 + 1 + 3 = 40

Make a note

- ✓ By default, modifier in Scala is public

3. Scala coding introduction

3.1 Ways to write Scala program

- ✓ We can write the Scala programs in different ways.
  - By using any text editor like Notepad++, Edit plus.
  - We can also write Scala programs by using Scala shell, REPL(read, eval, print, loop)
  - We can develop Scala programs by using any IDE(Integrated Development Environment) like Intellij, eclipse, etc...

3.2 Scala program execution steps

- ✓ We need to write Scala programs in notepad (good approach for practice initially).
- ✓ We can save the program with .scala (dot scala) extension.
- ✓ We need to compile the program
- ✓ Run or execute the program.
- ✓ Finally, we will get output.

3.3 Learn before touch scala program

- ✓ Compile and run Scala program
  - To compile we need to use scalac command
  - To Run or execute we need to use scala command

Make a note	Syntax to compile and run scala program
Compile	scalac filename
Run	scala classname
Compile	scalac Demo.scala
Run	scala Demo

3.4 Scala Hello World program

Program Name: Scala hello world program
Demo1.scala

```
object Demo1
{
        def main(args: Array[String])
        {
                println("Welcome to Scala world")
        }
}
```

Compile: scalac Demo1.scala
Run: scala Demo1

Output

Welcome to Scala world

3.5 Scala Program explanation

- ✓ object
  - It is keyword in scala
  - By using this we need to create main class in scala
  - This class is entry point to scala program
  - Regarding this keyword we will learn much in OOPs concept (Thanks for understand)
- ✓ Program execution starts from main(args: Array[String]) method and we should follow the same syntax for main method
- ✓ main() method is the entry point to execute the programs.
- ✓ args: Array[String], this is command line arguments (will learn in upcoming)
- ✓ println() is a predefined method to print any content on consol.

Discussion

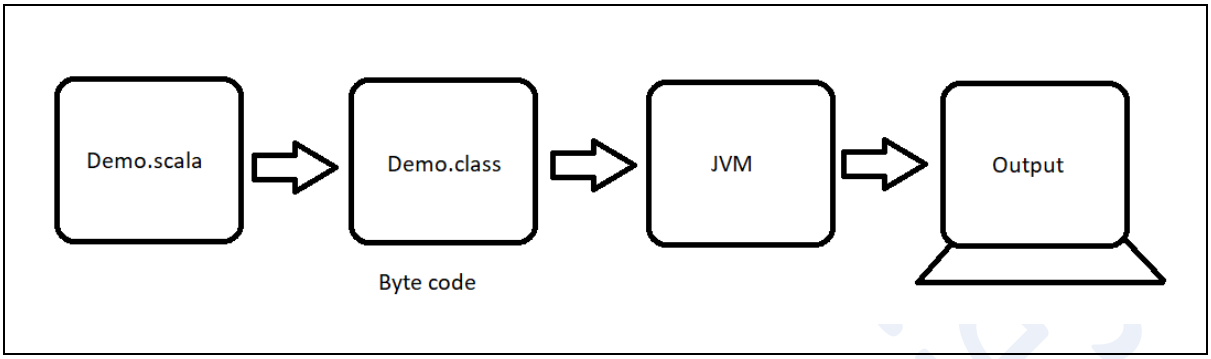
Nireekshan :

- Okay well Arjun sir, but internally what's happening when we run scala program?

Arjun :

- Yeah Nireekshan, good question, let me explain,
- To understand that we need to understand about scala internal program flow, please observe below one

3.5 Scala internal program flow



- ✓ In very first step we need to ***write*** a scala program
- ✓ The written scala program we need to save with **.scala** extension.
  - Example : Demo.scala
- ✓ We need to compile this program by using scalac command.
  - Example : scalac Demo.scala
- ✓ While compiling, compiler takes this source code and convert this file into corresponding **.class** file(s).
  - Example : Demo.class
- ✓ This **.class** file contains byte code instructions.
- ✓ These Byte code instructions cannot understandable by the microprocessor to generate output.
- ✓ So, the next step is we need to run or execute this program.
- ✓ To execute this program, we need to use scala command.
  - Example : scala Demo
- ✓ While running or executing the program ***JVM*** will take responsible to convert byte code instructions into machine understandable format.
- ✓ Then finally processor will generate output.
  - Welcome to scala world

Make a note

- ✓ We can also write scala program by following below syntax

3.6 We can also write like below program.

Program Name: Scala hello world program
Demo2.scala

```
object Demo2 extends App
{
        println("Welcome to Scala world")
}
```

Compile: scalac Demo2.scala
Run: scala Demo2

Output

```
Welcome to Scala world
```

Make a note

- ✓ Here ***App*** is a predefined class
- ✓ This class contains main method internally
- ✓ So, what is a class we will understand more in OOPs upcoming topic

4. Naming conventions in Scala

4.1 What is identifier?

- ✓ A name in a Scala program is called identifier.
- ✓ This name can be,
    - class name
    - package name
    - variable name
    - function name
    - method name
- ✓ Scala developers made some suggestions to the programmers regarding how to write identifiers in program.

4.2 Why should we follow naming conventions?

- ✓ If we follow the naming conventions, then the written code is,
    - Easy to understand.
    - Easy to read.
    - Easy to debug.

Make a note

- ✓ To execute all this chapter programs, I'm taking scala variable example.
- ✓ In scala we can create a variable by using var keyword
- ✓ Regarding variables we will learn more in 4th chapter

4.3 Rules to define identifiers in Scala:

1. The only allowed characters to write identifier in Scala are,
   - Alphabets, these can be either lower case or upper case.
   - Digits (0 to 9)
   - Below symbols
     - Underscore symbol (_)
     - Rupee symbol ($)

Program Name: checking naming convention
Demo1.scala

```
object Demo1
{
        def main(args: Array[String])
        {
                var studentId=101
                println("Student id is: "+studentId)
        }
}
```

Compile: scalac Demo1.scala
Run: scala Demo1

Output

```
Student id is:101
```

Make a note

- ✓ + Operator is concatenating string and variable value in println method.
- ✓ We will learn in variables chapter

2. Identifier allowed digits, but identifier should not start with digit.

Program Name: printing variable name which having digits in end of the name
Demo2.scala

```
object Demo2
{
        def main(args: Array[String])
        {
                var studentId123=101
                println("Student id is: "+studentId123)
        }
}
```

Compile: scalac Demo2.scala
Run: scala Demo2

Output

```
Student id is:101
```

Program Name: error: printing variable name which starts with digits, it is invalid
Demo3.scala

```
object Demo3
{
        def main(args: Array[String])
        {
                var 123studentId=101
                println("Student id is: "+123studentId123)
        }
}
```

Compile: scalac Demo3.scala
Run: scala Demo3

Error

```
Demo3.scala:5: error: Invalid literal number
                var 123studentId=101
                    ^
```

3. Identifiers are case sensitive.

Program Name: error: To prove scala is case sensitive
Demo4.scala

```
object Demo4
{
        def main(args: Array[String])
        {
                var a = 101
                println("Value of a is: "+A)
        }
}
```

Compile: scalac Demo4.scala
Run: scala Demo4

Error

```
Demo4.scala:6: error: not found: value A
                println("Student id is:"+A)
                                        ^
```

4. We cannot use keywords as identifiers.

Program Name: error: printing variable name given as keyword name, it is invalid
Demo5.scala

```
object Demo5
{
        def main(args: Array[String])
        {
                var if = 101
                println("Value of a is: "+if)
        }
}
```

Compile: scalac Demo5.scala
Run: scala Demo5

Error

```
Demo5.scala:5: error: illegal start of simple pattern
                var if=101
                   ^
```

5. Spaces are not allowed between identifier.

Program Name: error: space is not allowed while creating an identifier
Demo6.scala

```
object Demo6
{
        def main(args: Array[String])
        {
                var student id = 101
                println("Value of a is: "+student id)
        }
}
```

Compile: scalac Demo6.scala
Run: scala Demo6

Error

```
Demo6.scala:5: error: illegal start of simple pattern
                var student id=101
                              ^
```

Validate the below identifiers

- ✓ 435student # invalid
- ✓ student564 # valid
- ✓ student565info # valid
- ✓ $tudent # valid
- ✓ _student_info # valid
- ✓ var class = 10 # invalid

Make a note

- ✓ This below scala program identifiers table you can understand only during chapter discussion (like oops or functions, etc...).
- ✓ For the time being if you skip also no issue.

4.4. Scala program identifiers table

1. class 2. trait 3. object	✓ Names should start with upper case and remaining letters are in lower case. ✓ If name having multiple words, then every inner word should start with upper case letter. ✓ Example: Student, EmployeeInfo
4. variable 5. function 6. method	✓ Names should be in lower case. ✓ If name having multiple words then every inner word should start with upper case letter. ✓ Example: id, employeeNumber
8. package	✓ Name should starts with lower case letter. ✓ If name having multiple words then every inner word should start with lower case letter. ✓ Example: org, org.apache
9. Constants	✓ All letters in name should be capital letters only ✓ Example: CANCEL, RUNNING

4.6. Comments

- ✓ Comments are useful to describe about the code in an easy way.
- ✓ Scala compiler ignores comments while compiling the program.

4.6.1 Types of comments

- ✓ There are two types of comments in scala
  - Single comments
    - Single comments are useful to comment single lines in program
    - By using **//** two slashes symbol we will comment single lines
  - Multi comments
    - Multi comments are useful to comment multiple lines in program
    - By using /** this symbol we will comment multiple lines
    - /** by using this symbol we need to opening comments and by using **/ this symbol we need to close the multi comments

Program Name: Single comments in scala program
Demo7.scala

```
// This is basic scala program
object Demo7
{
        def main(args: Array[String])
        {
                println("Welcome to scala world")
        }
}
```

Compile: scalac Demo7.scala
Run: scala Demo7

Output

```
Welcome to scala world
```

Program Name: Multi comments in scala program
Demo8.scala

```
/** This program written by Arjun
at DVS during session delivering **/

object Demo8
{
        def main(args: Array[String])
        {
                println("Welcome to scala world")
        }
}
```

Compile: scalac Demo8.scala
Run: scala Demo8

Output

```
Welcome to scala world
```

5. Variables

5.1. Variable

- ✓ Variable means,
  - It's a name.
  - A variable refers to a value.
  - A variable holds the data
  - A variable is a name of the memory location.

Program Name	Creating a simple variable Demo1.scala

```
object Demo1
{
        def main(args: Array[String])
        {
                var studentId=101
                println(studentId)
        }
}
```

Compile	scalac Demo1.scala
Run	scala Demo1
Output	101

5.2. Properties of variable

- ✓ Every variable has a,
  - Type
  - Value
  - Scope
  - Location
  - Life time

5.3. Creating a variables

- ✓ Scala provides two keywords to create variables,
  - var
  - val
- ✓ So, to create a variable we need to specify,
  - The name of the variable with either var or val
  - Assign a value to name of the variable.
    - Here value also called as literal or constant

5.4. var keyword

- ✓ var is a keyword in scala programming language.
- ✓ By using var we can create a variable.
- ✓ var variable having mutable nature

5.5. What is mutable?

- ✓ Mutable means changing the nature
- ✓ Once if we create a variable by using var then we can change the var variable value.
  - Mutable variable can re-assign the existing variable value
  - Mutable variable can modify the existing variable value
  - Mutable variable can update the existing variable value

5.6. Creating variable by using var

var variable Syntax 1

```
var nameOfTheVariable = value
```

var variable Syntax 2

```
var nameOfTheVariable: Typeofvariable = value
```

Program Name: Creating variable by using var keyword
Demo2.scala

```
object Demo2
{
    def main(args: Array[String])
    {
        var age=16
        println(age)
    }
}
```

Compile: scalac Demo2.scala
Run: scala Demo2

Output

```
16
```

Program Name: Creating variable by using var keyword by using Data type
Demo3.scala

```
object Demo3
{
	def main(args: Array[String])
	{
		var age: Int=16
		println(age)
	}
}
```

Compile: scalac Demo3.scala
Run: scala Demo3

Output

```
16
```

Few points to make a note

- ✓ var is keyword
- ✓ Int is data type name
- ✓ **:** is separator between variable and data type

Make a Note

- ✓ We can print meaningful text message along with variable for better understanding
  - Text message we should keep in within double quotes.
  - Text message and variable name should be separated by plus symbol(+).
  - This symbol concatenates the message and variable values

Program Name: Creating variable by using var keyword
Demo4.scala

```
object Demo4
{
	def main(args: Array[String])
	{
		var age=16
		println("My age is sweet: "+age)
	}
}
```

Compile: scalac Demo4.scala
Run: scala Demo4

Output

```
My age is sweet: 16
```

Program Name	Creating variable and reassigning value Demo5.scala

```
object Demo5
{
        def main(args: Array[String])
        {
                var age=16
                age=18
                println(age)
        }
}
```

Compile	scalac Demo5.scala
Run	scala Demo5
Output	18

5.8. When should we go for var variable?

- ✓ In whole over application if the value of the variable is changing frequently then we should declare that variable with var.

5.9. Conclusion for var

- ✓ Re-assignment is possible if we create variable by using var keyword

7. val keyword

- ✓ val is a keyword in scala programming language.
- ✓ By using val we can create a variable.
- ✓ val variable having immutable nature

7.1. What is immutable?

- ✓ Immutable means we cannot change the existing nature.
- ✓ Once if we create a variable by using val then we cannot change the val variable value.
  - Immutable variable cannot re-assign the existing variable value
  - Immutable variable cannot modify the existing variable value
  - Immutable variable cannot update the existing variable value

7.2. Creating variable by using val

val variable Syntax 1

```
val nameOfTheVariable = value
```

val variable Syntax 2

```
val nameOfTheVariable: Typeofvariable = value
```

Program Name: Creating variable by using val keyword
Demo6.scala

```
object Demo6
{
	def main(args: Array[String])
	{
		val empId=101
		println(empId)
	}
}
```

Compile: scalac Demo6.scala
Run: scala Demo6

Output

```
101
```

Program Name: Creating variable by using val keyword with data type
Demo7.scala

```
object Demo7
{
        def main(args: Array[String])
        {
                val empId: Int=101
                println(empId)
        }
}
```

Compile: scalac Demo7.scala
Run: scala Demo7

Output

```
101
```

Few points to make a note

- ✓ val is keyword
- ✓ Int is data type name
- ✓ **:** is separator between variable and data type

Make a Note

- ✓ We can print meaningful text message along with variable for better understanding
  - Text message we should keep in within double quotes.
  - Text message and variable name should be separated by plus symbol(+).
  - This symbol concatenates the message and variable values

Program Name: Creating variable by using val keyword
Demo8.scala

```
object Demo8
{
        def main(args: Array[String])
        {
                val empId =16
                println("My employee id is:"+ empId)
        }
}
```

Compile: scalac Demo8.scala
Run: scala Demo8

Output

```
My employee id is: 101
```

Program Name: error: We cannot reassign values to existing val variables values
Demo9.scala

```
object Demo9
{
        def main(args: Array[String])
        {
                val empId =101
                empId =111
                println(empId)
        }
}
```

Compile: scalac Demo9.scala
Run: scala Demo9

Error

```
Demo9.scala:6: error: reassignment to val
                empId =111
                    ^
```

7.4. When should we go for val variable?

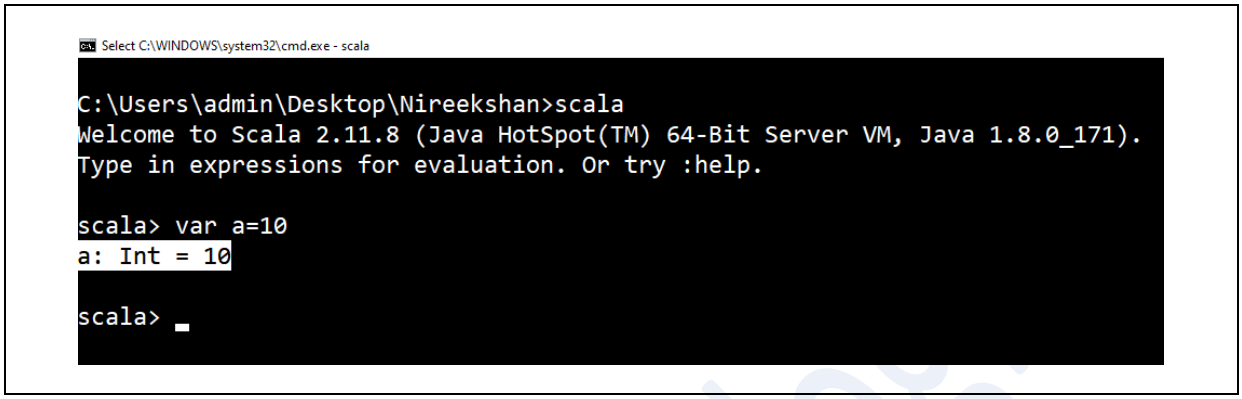
- ✓ In whole over application if the value of the variable is not changing frequently then we should declare that variable with val.

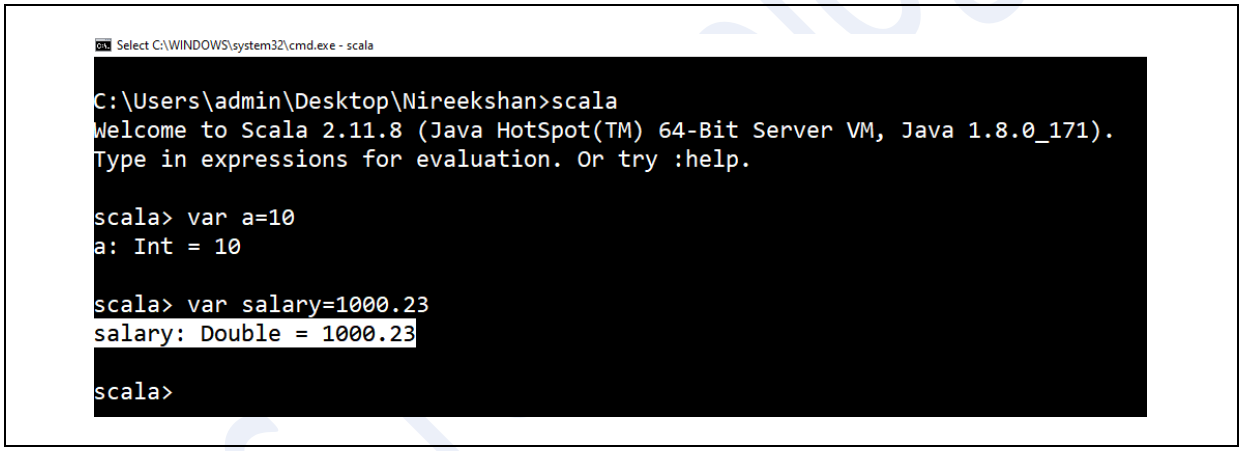
7.5. Conclusion for val

- ✓ Re-assignment is not possible if we create variable by using val keyword

8. Type inference

- ✓ If we didn't provide the type of value, then scala interpreter provides the type this is called as type inference.
- ✓ We can check in scala REPL





9. null value

- ✓ null is a keyword in scala programming language.
- ✓ While creating a variable we can assign a value as null
- ✓ null value of the variable indicates as that variable or object is empty means nothing

Program Name: Creating variable and assigning with null value
Demo10.scala

```
object Demo10
{
        def main(args: Array[String])
        {
                val a=null

                println("This variable is holding null value: "+a)
        }
}
```

Compile: scalac Demo10.scala
Run: scala Demo10

Output

```
This variable is holding null value: null
```

Make a note

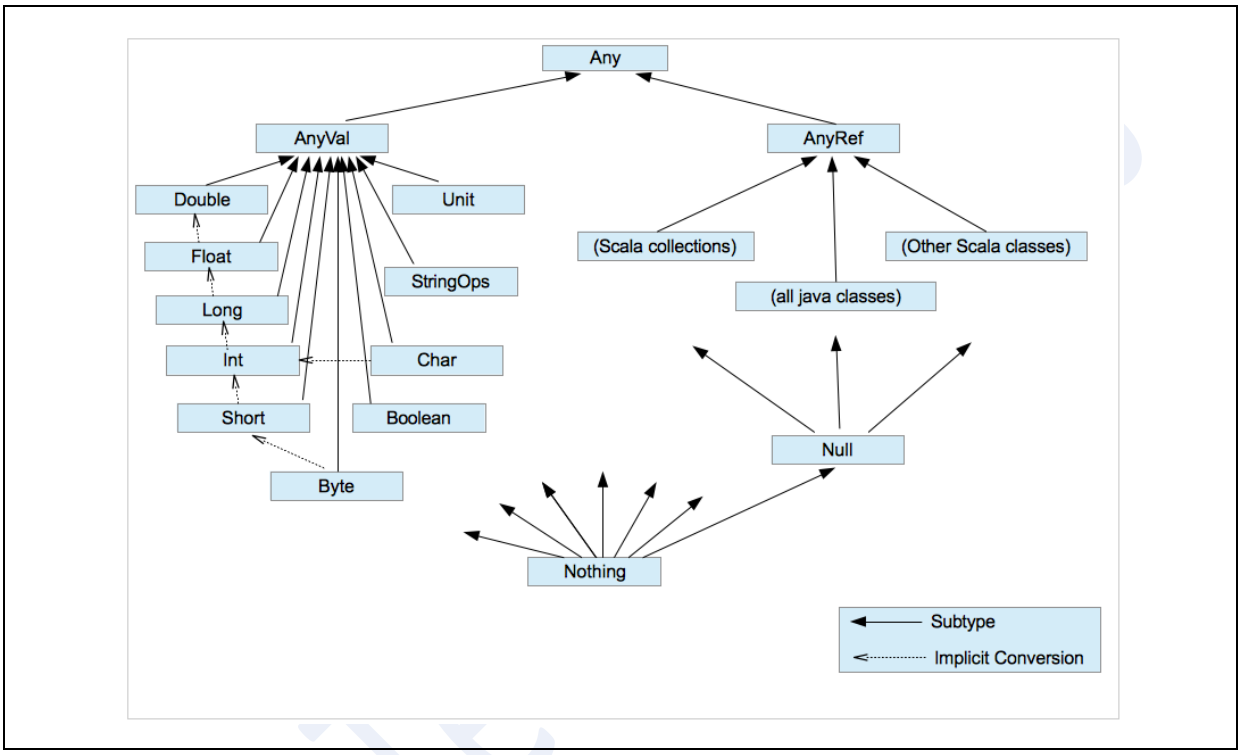
- ✓ In functional programming language variables are two types,
  - global variables
  - local variables
- ✓ In Object oriented programming variables are three types,
  - Instance variables
  - Singleton variables
  - Local variables
- ✓ These variables discussion we will have in functions and oops

6. Data types in scala

1. What is a data type?

- ✓ A data type represents the type of the data stored into a variable or memory.

2. Scala data type's image



3. Data type

- ✓ A data type represents the type of the data.
- ✓ In scala all a data types are predefined classes.
- ✓ Any is a predefined class in scala
  - AnyVal and AnyRef are child classes to Any class
- ✓ Regarding what is a class we will learn in OOPs upcoming chapter.

Program Name: Creating a simple variable
Demo1.scala

```
object Demo1
{
        def main(args: Array[String])
        {
                val x: Int = 10
                println(x)
        }
}
```

Compile: scalac Demo1.scala
Run: scala Demo1

Output

```
10
```

- ✓ Here a is Int means integer type of data
- ✓ Int is a predefined data type in scala

4. What is the default package in scala?

- ✓ Default package in scala is, scala package.
- ✓ Explicitly we no need to import this package in program.
- ✓ Automatically this package will be import

5. Mostly usage classes from scala package

Integral data types

1. scala.Byte
2. scala.Short
3. scala.Int
4. scala.Long

Floating data types

5. scala.Float
6. scala.Double

Character data type

7. scala.Char

Boolean data type

8. scala.Boolean

6. Types of data types

- ✓ There are mainly four types of data types.

1. Numeric data types
    1. Integral data types
        1. Byte
        2. Short
        3. Int
        4. Long
    2. Floating data types
        1. Float
        2. Double
2. Char data type
    1. Char
3. Boolean data type
    1. Boolean

6. 1. Numeric data types

- ✓ These data types represent number without decimal point.
- ✓ By default, data type for Integral data type is Int

1. Integral data types

    1. Byte
    2. Short
    3. Int
    4. Long

Data type	Memory size	Min and Max
1. Byte	1 byte (8 bits)	- 128 to +127
2. Short	2 bytes (16 bits)	- 32768 to +32767
3. Int	4 bytes (32 bits)	- 2147483648 to + 2147483647
4. Long	8 bytes (64 bits)	- 2 to the power 63 to + 2 to the power 63 -1

6.1.1. Byte data type

Size	:	1 byte
Min	:	- 128
Max	:	+ 127
Range	:	- 128 to + 127

Program Name: Creating Byte data type variable
Demo2.scala

```
object Demo2
{
	def main(args: Array[String])
	{
		val a: Byte = 10
		print(a)
	}
}
```

Compile: scalac Demo2.scala
Run: scala Demo2

Output:

```
10
```

Examples

```
val a: Byte = 10              // valid
val b: Byte = 130             // Error:    type mismatch;
val c: Byte = 10.5            // Error:    type mismatch;
val d: Byte = true            // Error:    type mismatch;
val e: Byte = "spark"         // Error:    type mismatch;
```

6.1.2. Short

Size	:	2 bytes
Min	:	- 32768
Max	:	+ 32767
Range	:	- 32768 to + 32767

Program Name: Creating Short data type variable
Demo3.scala

```
object Demo3
{
	def main(args: Array[String])
	{
		val a: Short = 10000
		print(a)
	}
}
```

Compile: scalac Demo3.scala
Run: scala Demo3

Output:

```
10000
```

Examples

```
val a: Short = 10              // valid
val b: Short = 32769           // Error:    type mismatch;
val c: Short = 10.5            // Error:    type mismatch;
val d: Short = true            // Error:    type mismatch;
val e: Short = "spark"         // Error:    type mismatch;
```

6.1.3. Int

Size	:	4 bytes
Min	:	-2147483648
Max	:	+ 2147483647
Range	:	- 2147483648 to + 2147483647

Program Name: Creating Int data type variable
Demo4.scala

```
object Demo4
{
	def main(args: Array[String])
	{
		val a: Int = 10000
		print(a)
	}
}
```

Compile: scalac Demo4.scala
Run: scala Demo4

Output:

```
10000
```

Examples

```
val a: Int = 10                 // valid
val b: Int = 2147483649         // Error:   integer number too large
val c: Int = 10.5               // Error:   type mismatch;
val d: Int = true               // Error:   type mismatch;
val e: Int = "spark"            // Error:   type mismatch;
```

6.1.4. Long

Size : 8 bytes

Program Name: Creating Long data type variable
Demo5.scala

```
object Demo5
{
	def main(args: Array[String])
	{
		val a: Long = 10000
		print(a)
	}
}
```

Compile: scalac Demo5.scala
Run: scala Demo5

Output:

```
10000
```

Examples

```
val a: Long = 10                // valid
val b: Long = 10.5              // Error:   type mismatch;
val c: Long = true              // Error:   type mismatch;
val d: Long = "spark"           // Error:   type mismatch;
```

6.2. Floating Point Data types:

- ✓ These data types represent the numbers with decimal point.
- ✓ By default, data type for Floating data type is Double

- ✓ Floating data types

1. Float
2. Double

Data type	Memory size	Min and Max
1. Float	4 bytes (8 bits)	-3.4e38 to +3.4e38
2. Double	8 bytes (16 bits)	-1.7e308 to +1.7e308

6.2.1. Float

- ✓ Floating value should be prefix with **f**

Size : 4 bytes

Program Name: Creating Float data type variable
Demo6.scala

```
object Demo6
{
	def main(args: Array[String])
	{
		val a: Float = 10000f
		print(a)
	}
}
```

Compile: scalac Demo6.scala
Run: scala Demo6

Output:

```
10000.0
```

Examples

```
val a: Float = 10.3f          // valid
val b: Float = 10.3           // Error:   type mismatch;
val c: Float = true           // Error:   type mismatch;
val d: Float = "spark"        // Error:   type mismatch;
```

6.2.2. Double

Size : 8 bytes

Program Name: Creating Double data type variable
Demo7.scala

```
object Demo7
{
	def main(args: Array[String])
	{
		val a: Double = 10000
		print(a)
	}
}
```

Compile: scalac Demo7.scala
Run: scala Demo7

Output:

```
10000
```

Examples

```
val a: Double = 10.3          // valid
val d: Double = true          // Error:    type mismatch;
val e: Double = "spark"       // Error:    type mismatch;
```

6.2.3. Char Data types

Size	:	2 bytes
Min	:	0
Max	:	+ 65535
Range	:	0 to + 65535

- ✓ Character data means it's a single letter.
- ✓ A single character is enclosed within the single quotes.

Program Name: Creating Char data type variable
Demo8.scala

```
object Demo8
{
        def main(args: Array[String])
        {
                val a: Char = 'm'
                print(a)
        }
}
```

Compile: scalac Demo8.scala
Run: scala Demo8

Output:

```
m
```

Examples

```
val a: Char = 'a'          // valid
val a: Char = 'A'          // valid
val b: Char = 99           // valid
val c: Char = 'abc'        // Error:   unclosed character literal
val e: Char = "spark"      // Error:   type mismatch;
```

6.2.4. Boolean Data types:

- ✓ The allowed values for Boolean data type are true and false.
- ✓ We can use Boolean data type to represent logical values.

Program Name: Creating Boolean data type variable
Demo9.scala

```
object Demo9
{
    def main(args: Array[String])
    {
        val a: Boolean = true
        print(a)
    }
}
```

Compile: scalac Demo9.scala
Run: scalaDemo9

Output:

```
true
```

Examples

```
val a: Boolean = true          // valid
val a: Boolean = false         // valid
val b: Boolean = 130           // Error:   type mismatch;
val c: Boolean = 10.5          // Error:   type mismatch;
val e: Boolean = "spark"       // Error:   type mismatch;
```

7. Summary

- ✓ By default scala package name is scala
  - Byte, Short, Int, Long, Float, Double, Char, Boolean are predefined classes available in scala package

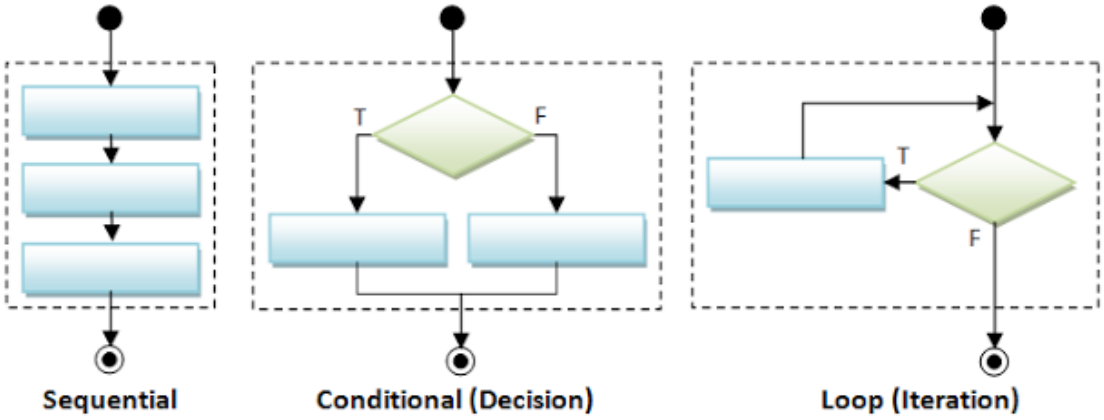
7. Flow control

1. Why should we learn about flow control?

- ✓ Simple answer: To understand the flow of statements execution in a program.
- ✓ In any programming language, statements will be executed mainly in three ways,
  - Sequential.
  - Conditional.
  - Looping.

2. Flow control

- ✓ The order of statements execution is called as flow of control.
- ✓ Based on requirement the programs statements can executes in different ways like sequentially, conditionally and repeatedly etc.



2.1. Sequential

- ✓ Statements execute from top to bottom, means one by one sequentially.
- ✓ By using sequential statement, we can develop only simple programs.

2.2. Conditional

- ✓ Based on the conditions, statements used to execute.
- ✓ Conditional statements are useful to develop better and complex programs.

2.3. Looping

- ✓ Based on the conditions, statements used to execute randomly and repeatedly.
- ✓ Looping execution is useful to develop better and complex programs.

2.1.1. Sequential statements

- ✓ Statements will execute from top to bottom, means one by one

Program Name: Creating variable by using val keyword
Demo1.scala

```
object Demo1
{
        def main (args: Array[String])
        {
                println("one")
                println("two")
                println("three")
                println("four")
        }
}
```

Compile: scalac Demo1.scala
Run: scala Demo1

Output

```
one
two
three
four
```

3. Conditional or Decision-making statements

- ✓ if
- ✓ if else
- ✓ if else if
- ✓ match

4. Looping

- ✓ while
- ✓ do while
- ✓ for

5. others

- ✓ return

6. *if* statement

```
syntax

        if(expression/condition)
        {
                statements
        }
```

- ✓ if statement holds an expression.
- ✓ Expression gives the result as Boolean type means either true or false.

- ✓ If the result is *true*, then if block statements will execute.
- ✓ If the result is *false*, then if block statements will not execute.

6.1. When should we use if statement?

- ✓ If you want to do either one thing or nothing at all then you should go for if statement.

Program Name: Basic program on if statement
Dem2.scala

```
object Demo2
{
        def main(args: Array[String])
        {
                val a: Int = 10

                println("value of (a==10) is "+(a == 10))

                if(a == 10)
                {
                        println("a value is 10")
                }
        }
}
```

Compile: scalac Demo2.scala
Run: scala Demo2

output

```
value of (a==10) is true
a value is 10
```

Program Name: Basic program on if statement
Demo3.scala

```
object Demo3
{
        def main(args: Array[String])
        {
                val a: Int = 10

                println("value of (a==20) is  "+(a == 20))

                if(a == 20)
                {
                        println("a value is 10")
                }
        }
}
```

Compile: scalac Demo3.scala
Run: scala Demo3

output

```
value of (a==20) is false
```

7. *if else* statement

```
syntax

        if(expression/condition)
        {
                statements
        }

        else
        {
                statements
        }
```

- ✓ If statement holds an expression.
- ✓ Expression gives the result as boolean type means either true or false.

- ✓ If the result is *true*, then if block statements will execute
- ✓ If the result is *false*, then else block statements will execute.

7.1. When should we use if statement?

- ✓ If you want to do either one thing or another thing then you should go for if else statement.

Program Name: Basic program on if else statement
Demo4.scala

```
object Demo4
{
	def main(args: Array[String])
	{
		val hour: Int = 12

		println("value of (hour<=12) is: "+(hour == 12))

		if(hour <= 12)
		{
			println("Good morning")
		}

		else
		{
			println("I'm sure it is not morning")
		}
	}
}
```

Compile: scalac Demo4.scala
Run: scalaDemo4

output

```
value of (hour<=12) is: true
Good morning
```

Program Name: Basic program on if else statement
Demo5.scala

```
object Demo5
{
	def main(args: Array[String])
	{
		val hour: Int = 20

		println("value of (hour<=12) is: "+(hour == 12))

		if(hour <= 12)
		{
			println("Good morning")
		}

		else
		{
			println("I am sure it is not morning")
		}
	}
}
```

Compile
scalac Demo5.scala

Run
scala Demo5

output

```
value of (hour<=12) is: false
I am sure it is not morning
```

8. if else if statement

syntax

```
if(expression/condition)
{
        statements
}

else if(expression/condition)
{
        statements
}

else if(expression/condition)
{
        statements
}

else
{
        statements
}
```

- ✓ If and else-if statements holds an expression.
- ✓ Expression gives the result as boolean type means either true or false.

- ✓ If the result is *true*, then any matched if or else if block statements will execute
- ✓ If the result is *false*, then else block statements will execute.

8.1 When should we use if statement?

- ✓ This we can use to choose a option from more than two possibilities.

Program Name: Basic program on if else if statement
Demo6.scala

```
object Demo6
{
	def main(args: Array[String])
	{
		val marks: Int = 60

		if(marks >= 90)
		{
			println("A grade")
		}

		else if(marks >= 80)
		{
			println("B grade")
		}

		else if(marks >= 70)
		{
			println("C grade")
		}

		else if(marks >= 60)
		{
			println("D grade")
		}

		else if(marks >= 35)
		{
			println("E grade")
		}

		else
		{
			println("Fail")
		}
	}
}
```

Compile: scalac Demo6.scala
Run: scala Demo6

Output

```
D grade
```

9. Summary

if	Select one solution or nothing
if else	Select either one solution or another solution
if else if	Select one solution from multiple solutions

10. Looping

3.1 do while
3.2 while
3.3 for

11. do while

Syntax

```
initialization

do
{
        statements
        increment

} while(expression/condition)
```

- ✓ do while loop holds expression
- ✓ Expression gives the result as boolean type means either true or false.

- ✓ If the result is true, then do while loop executes till condition reaches to false
- ✓ If the result is false, then do while loop terminates.
- ✓ As per the syntax, the checking of expression will be done after the code got executed.
- ✓ So, do while loop will execute at least one time even though if the condition returns false.

Program Name: Print 1 to 5 by using do while loop
Demo7.scala

```
object Demo7
{
	def main(args: Array[String])
	{
		var counter = 1

		do
		{
			println(counter)
			counter = counter + 1

		} while(counter<=5)
	}
}
```

Compile: scalac Demo7.scala
Run: scala Demo7

Output

```
1
2
3
4
5
```

Program Name: do while loop executes once even condition fails
Demo8.scala

```
object Demo8
{
	def main(args: Array[String])
	{
		var counter = 1

		do
		{
			println(counter)
			counter = counter + 1

		} while(counter>=5)
	}
}
```

Compile: scalac Demo8.scala
Run: scala Demo8

Output

```
1
```

12. while

Syntax

```
Initialization

while(expression/condition)
{
        statements
        increment/decrement
}
```

- ✓ While loop holds expression
- ✓ Expression gives the result as Boolean type means either true or false.

- ✓ If the result is true, then while loop executes till condition reaches to false
- ✓ If the result is false, then while loop terminates.
- ✓ As per while loop syntax, the checking of expression will be done at first only.
- ✓ So, if expression returns false then it displays nothing.

Program Name: Print 1 to 5 by using while loop
Demo9.scala

```
object Demo9
{
	def main(args: Array[String])
	{
		var counter = 1

		while(counter<=5)
		{
			println(counter)
			counter = counter + 1
		}
	}
}
```

Compile: scalacDemo9.scala
Run: scala Demo9

output

```
1
2
3
4
5
```

Program Name: while loop won't execute initially if condition false
Demo10.scala

```
object Demo10
{
	def main(args: Array[String])
	{
		var counter = 1

		while(counter>=5)
		{
			println(counter)
			counter = counter + 1
		}
	}
}
```

Compile: scalac Demo10.scala
Run: scala Demo10

output

13. for loop (for *comprehension* or *for expression*)

- ✓ for loop used to iterate or get one by one object from collection object.
- ✓ It is also used to filter and return an iterated collection.
- ✓ for loop also called as for-comprehension
- ✓ for works with many combinations

- for - to
- for - until
- for - by
- for - yield

Syntax

```
for (i <- start to end)
{
        statements to execute
}
```

Make a note

- ✓ This symbol <- is called as generator

Program Name: Example using for loop
Demo11.scala

```
object Demo11
{
        def main(args: Array[String])
        {
                for(i <- 1 to 5)
                {
                        println(i)
                }
        }
}
```

Compile: scalac Demo11.scala
Run: scalaDemo11

output

```
1
2
3
4
5
```

Syntax

```
for (i <- start until end)
{
        statements to execute
}
```

14. Difference between until and to

- ✓ to : It includes start and end value given in the range
- ✓ until : It excludes last value of the range

Program Name: Example using for loop
Demo12.scala

```
object Demo12
{
        def main(args: Array[String])
        {
                for(i <- 1 until 5)
                {
                        println(i)
                }
        }
}
```

Compile: scalac Demo12.scala
Run: scala Demo12

output

```
1
2
3
4
```

15. Scala for-loop example using by keyword

- ✓ for with by is using to skip the iteration.
- ✓ When you code like by 2 it means, this loop will skip all even iterations of loop.

Program Name: Example using for loop
Demo13.scala

```
object Demo13
{
        def main(args: Array[String])
        {
                for(i<-1 to 10 by 2)
                {
                        println(i)
                }
        }
}
```

Compile: scalac Demo13.scala
Run: scalaDemo13

output

```
1
3
5
7
9
```

16. Scala for-loop filtering example

- ✓ We can use for loop to filter the data
- ✓ Based on condition we can filter the data or values.

Program Name: Example using for loop
Demo14.scala

```
object Demo14
{
        def main(args: Array[String])
        {
                for( a<- 1 to 10 if a%2==0 )
                {
                        println(a)
                }

        }
}
```

Compile: scalac Demo14.scala
Run: scalaDemo14

output

```
2
4
6
8
10
```

17. Scala for-loop example by using yield keyword

- ✓ In scala, for loop with yield keyword combination is valid.
- ✓ For with yield loop returns a collection object.
- ✓ Internally for loop uses buffer memory to store each iteration result.
- ✓ Once all iterations done this buffer memory returns the result.

- ✓ If for and yield works with Array, then it returns Array object
- ✓ If for and yield works with Map, then it returns Map object
- ✓ If for and yield works with List, then it returns List object

Program Name: Example using for loop
Demo15.scala

```
object Demo15
{
        def main(args: Array[String])
        {
                var result = for( a<- 1 to 5) yield a

                for(i<-result)
                {
                        println(i)
                }
        }
}
```

Compile: scalac Demo15.scala
Run: scala Demo15

output

```
1
2
3
4
5
```

18. Scala for-loop in Collection

Program Name: Example using for loop
Demo16.scala

```
object Demo16
{
        def main(args: Array[String])
        {
                var list = List(1,2,3,4,5)

                for( i<- list)
                {
                        println(i)
                }
        }
}
```

Compile: scalac Demo16.scala
Run: scala Demo16

output

```
1
2
3
4
5
```

match and case keywords(Pattern matching)

- ✓ Match and case are keywords in scala programming language.
- ✓ This match and case combination also called as pattern matching.
- ✓ It is the most widely used feature in Scala.
- ✓ It is a technique for checking a value against a pattern.
- ✓ It is like a *switch statement of Java* and *C*.
- ✓ Here, "**match**" keyword is used instead of switch statement.
- ✓ Syntactically match keyword may contain several cases.
- ✓ Each case statement includes a pattern and one or more expression.
- ✓ Pattern and case will be separated by arrow symbol(=>)

Program Name: Pattern matching example
Demo17.scala

```
object Demo17
{
        def main(args: Array[String])
        {
                var a = 1

                a match
                {
                        case 1 => println("One")
                        case 2 => println("Two")
                        case _ => println("No")
                }
        }
}
```

Compile: scalac Demo17.scala
Run: scala Demo17

Output

```
One
```

Make a note

- ✓ Here, match using a variable named a.
- ✓ This variable matches with best available case and print output.
- ✓ Underscore (_) is used in the last case for making it default case, means if nothing matches then default case will executes.

Program Name: Pattern matching example
Demo18.scala

```
object Demo18
{
        def main(args: Array[String])
        {
                var a = 11

                a match
                {
                        case 1 => println("One")
                        case 2 => println("Two")
                        case _ => println("No")
                }
      }
}
```

Compile: scalac Demo18.scala
Run: scala Demo18

Output

```
No
```

Pattern matching with functions

- ✓ We can apply pattern matching with function concept

Program Name: Pattern matching using with function example
Demo19.scala

```
object Demo19
{
        def main(args: Array[String])
        {
                var result = search ("Hello")
                print(result)
        }

        def search (a:Any):Any = a match
        {
                case 1 => println("One")
                case "Two" => println("Two")
                case "Hello" => println("Hello")
                case _ => println("No")
      }
}
```

Compile: scalac Demo19.scala
Run: scala Demo19

Output

```
Hello
()
```

Make a note

- ✓ Patten matching, we can use with case classes as well.
- ✓ Regarding what is a case class we will learn in oops concept.

8. Scala String

1. What is a String?

- ✓ A String represents a group of characters enclosed within double quotes.
- ✓ Scala depends on Java String.
- ✓ String is a pre-defined class presents in java.lang package

2. How to create String object?

There are two ways to create String object:

1. By string literal
2. By new keyword

2.1. String literal.

- ✓ We can create String by using String literal in scala
- ✓ String literal means by using double quotes we can create

Program Name	Creating string by using String literal Demo1.scala

```
object Demo1
{
        def main(args: Array[String])
        {
                var name="Nireekshan"
                println(name)
        }
}
```

Compile	scalac Demo1.scala
Run	scala Demo1
Output	Nireekshan

2.2. new keyword.

- ✓ We can create String object by using new keywords in scala

Program Name: checking naming convention
Demo2.scala

```
object Demo2
{
        def main(args: Array[String])
        {
                var name= new String("Nireekshan")
                println(name)
        }
}
```

Compile: scalac Demo2.scala
Run: scala Demo2

Output:

```
Nireekshan
```

3. Methods in String class

- ✓ Generally **String** is a predefined class
- ✓ Inside class methods should exists.
- ✓ So, String class can contain method to perform required operations

Make a note

- ✓ Methods are two types in Scala
  - Instance methods
  - Singleton methods
- ✓ Instance methods we should access by using object name
- ✓ Singleton methods we should access by using Singleton class name
- ✓ Inside String class all are instance methods
  - So, to access String class methods we need to create an object

Important methods for String class:

1. public int length()

- ✓ This method gives number of character present in String object.

Program Name: String length example
Demo3.scala

```
object Demo3
{
        def main(args: Array[String])
        {
                var s1= "abcdefg"

                println(s1.length())

        }
}
```

Compile: scalac Demo3.scala
Run: scala Demo3

Output

```
7
```

2. public String toLowerCase()

- ✓ This method converts all characters of the string onto lower case, and returns that lower cased string

Program Name: String toLowerCase example
Demo4.scala

```
object Demo4
{
        def main(args: Array[String])
        {
                var s1= "ABCDEFG"

                println(s1.toLowerCase())
        }
}
```

Compile: scalac Demo4.scala
Run: scala Demo4

Output:

```
abcdefg
```

3. public String toUpperCase()

- ✓ This method converts all characters of the string onto upper case, and returns that upper cased string

Program Name: String toUpperCase example
Demo5.scala

```
object Demo5
{
        def main(args: Array[String])
        {
                var s1= "abcdefg"

                println(s1.toUpperCase())
        }
}
```

Compile: scalac Demo5.scala
Run: scala Demo5

Output:

```
ABCDEFG
```

4. public String concat(String s)

- ✓ This method concatenates or joins two Strings and returns a third string as a result.

Program Name: String concat example
Demo6.scala

```
object Demo6
{
        def main(args: Array[String])
        {
                var s1= "Java"
                var s2= "Scala"

                println(s1.concat(s2))
        }
}
```

Compile: scalac Demo6.scala
Run: scala Demo6

Output

```
JavaScala
```

5. public boolean equals(Object obj)

- ✓ This method compares the content of two String objects then it returns the boolean as a result, where the case is also important.

Program Name: String equal example
Demo7.scala

```
object Demo7
{
        def main(args: Array[String])
        {
                var s1= "Java"
                var s2= "Java"
                var s3= "Scala"

                println(s1.equals(s2))
                println(s1.equals(s3))
        }
}
```

Compile: scalac Demo7.scala
Run: scala Demo7

Output

```
true
false
```

Program Name: String equal example
Demo8.scala

```
object Demo8
{
        def main(args: Array[String])
        {
                var s1= "Java"
                var s2= "java"
                var s3= "Scala"

                println(s1.equals(s2))
                println(s1.equals(s3))
                println(s1.equals(s2))
        }
}
```

Compile: scalac Demo8.scala
Run: scala Demo8

Output

```
false
false
false
```

6. public String substring(int begin)

- ✓ This method returns the substring from begin index to end-1 of the String.

Program Name: String substring example
Demo9.scala

```
object Demo9
{
        def main(args: Array[String])
        {
                var s1= "Java"

                println(s1.substring(1))
        }
}
```

Compile: scalac Demo9.scala
Run: scala Demo9

Output

```
ava
```

Program Name: String substring example
Demo10.scala

```
object Demo10
{
        def main(args: Array[String])
        {
                var s1= "Java"

                println(s1.substring(2))
        }
}
```

Compile: scalac Demo10.scala
Run: scala Demo10

Output

```
va
```

String Interpolation

- ✓ String interpolation means, adding variable references to string literals.

Program Name: String without interpolation
Demo11.scala

```
object Demo11
{
	def main(args: Array[String])
	{
		var name = "Nireekshan"

		println("I am "+name)
	}
}
```

Compile: scalac Demo11.scala
Run: scala Demo11

Output:

```
I am Nireekshan
```

Make a note

- ✓ In string interpolation concept we no need to use + operator to format your output string
- ✓ A string variable we can pass directly in string literal.

Program Name: String with interpolation
Demo12.scala

```
object Demo12
{
	def main(args: Array[String])
	{
		var name = "Nireekshan"

		println(s"I am $name")
	}
}
```

Compile: scalac Demo12.scala
Run: scala Demo12

Output:

```
I am Nireekshan
```

f interpolation to format numbers

- ✓ As an example, let's first print the price of one donut using the s interpolation
- ✓ f interpolations helps to print including decimal float vlaues

Program Name: String with interpolation
Demo12.scala

```
object Demo12
{
	def main(args: Array[String])
	{
		var x = 123.400

		println(s"x value is $x")
	}
}
```

Compile: scalac Demo12.scala
Run: scala Demo12

Output:

```
x value is 123.4
```

Program Name: String with interpolation
Demo12.scala

```
object Demo12
{
	def main(args: Array[String])
	{
		var x = 123.400

		println(f"x value is $x")
	}
}
```

Compile: scalac Demo12.scala
Run: scala Demo12

Output:

```
x value is 123.400
```

Object Oriented Programming System

9. OOPS - Part - 1 - class, object, methods, etc

Full form of OOPS

- ✓ The full for of OOPS is "Object Oriented Programming System"
- ✓ Scala is pure Object-Oriented Programming language.
  - Scala represents everything is an object.

What is OOPS exactly?

- ✓ It's a methodology to design software using classes and objects.

Why should we use?

- ✓ It simplifies the software development by providing oops features.

OOPS features

- class
- object
- Data binding
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism etc.

1). class

Definition 1:

- ✓ A class is a specification (idea/plan/theory) of properties and actions of objects.

Definition 2:

- ✓ A class is a model for creating objects and it does not exist physically.

Syntax

```
class NameOfTheClass
{
        1. constructor(s)
        2. variables (data members)
        3. methods (actions)
}
```

class keyword

- ✓ class is a keyword in scala programming language
- ✓ We can create a class by using class keyword

Inside class what we can define?

- ✓ class can contain mainly three parts,
    - constructor(s)
    - variables
    - methods

Hey Nireekshan, what is the purpose of constructor(s), variables and methods?

- ✓ Yeah Good question Boss,
    - Constructor purpose is to initialize instance variables
    - Variables purpose is to represent data
    - Methods purpose is to perform operations

Class naming convention

- ✓ class names should start with upper case and remaining letters are in lower case.
- ✓ If class name having multiple words, then every inner word should start with upper case letter.
- ✓ Examples:
  - Student
  - EmployeeInfo

Make a note

- ✓ If you did not follow naming convention, then you will not get any error.
- ✓ But its highly recommended to follow to meet real time coding standards

Validate below names

- Student - valid and highly recommended
- student - valid but not recommended
- EmployeeInfo - valid and highly recommended
- empoyeeinfo - valid but not recommended

Program Name: Create a Student class with variables and method
Demo1.scala

```
class Student
{
        var id: Int = 10
        var name: String = "Nireekshan"

        def display()
        {
                println("Student id is: "+id)
                println("Student name is : "+name)
        }
}

object Demo1
{
        def main(args: Array[String])
        {
                println("Welcome to oops session")
        }
}
```

Compile: scalac Demo1.scala
Run: scala Demo1

Output

```
Welcome to oops session
```

Explanation about Demo1.scala

- ✓ Created Student class
- ✓ Inside Student class created two variables and one method
- ✓ Created one standalone class.
- ✓ Inside standalone class created main method

Info:

- ✓ Boss writing a class is not enough; we should learn how to access variables and methods.

How to access?

- ✓ Simple and beautiful answer is,
  - We should create an object to a class.

2). object

Info

- ✓ Please don't get confuse between,
    - object keyword
    - Creating object to a class.
- ✓ Now we are discussing about creating object to class.

Then what is object keyword?

- ✓ In scala object keyword, by using object keyword we can create singleton class.
- ✓ Please hold your anxiety; we will learn full details about singleton class in upcoming chapter.
- ✓ Then let us start discussion about creating object to a class

Why should we create object for a class?

- ✓ Generally inside class we are defining variables and methods right.
- ✓ When we create an object to a class then only memory will be allocated to these variables and methods.
- ✓ So, hope you guys understand why we should create an object.
- ✓ Any questions the please…

What is an object?

Definition 1

- ✓ Instance of a class is known as an object.
- ✓ Instance
  - It is a mechanism of allocating memory space for data members of a class

Definition 2

- ✓ Grouped item is known as an object.
  - Object is a simple variable.
  - This variable holds group of data.

Definition 3:

- ✓ Logical runtime entities are called as objects.

Definition 4:

- ✓ Real world entities are called as objects.

Syntax 1:

```
val nameOfTheObject = new <NameOfTheClass>()
```

- ✓ We can create object for a class.
- ✓ We can create object by using new keyword

- ✓ nameOfTheObject --> This is an object name
- ✓ NameOfTheClass() --> This part is called as constructor.

- ✓ Regarding constructor we will learn in upcoming chapter.

Program Name: Create a Student class and object
Demo2.scala

```
class Student
{
        var id: Int = 101
        var name: String = "Nireekshan"

        def display()
        {
                println("Student id is: "+id)
                println("Student name is : "+name)
        }
}

object Demo2
{
        def main (args: Array[String])
        {
                println("Welcome to oops session")
                val s = new Student()
        }
}
```

Compile: scalac Demo2.scala
Run: scala Demo2

Output

```
Welcome to oops session
```

- ✓ Above program we have successfully created object
- ✓ Once after we create an object then happily we can access variable and methods

Program Name: Create a Student class and object to access variables and method
Demo3.scala

```
class Student
{
        var id: Int = 101
        var name: String = "Nireekshan"

        def display()
        {
                println("Student id is: "+id)
                println("Student name is : "+name)
        }
}


object Demo3
{
        def main (args: Array[String])
        {
                val s = new Student()
                s.display()
        }
}
```

Compile: scalac Demo3.scala
Run: scala Demo3

Output

```
Student id is: 101
Student name is: Nireekshan
```

Prasad

- ✓ Hey Nireekshan, can I create more than on object

Nireekshan

- ✓ Yes, Prasad we can create any number of objects for a class
- ✓ Make sure before creating object class should exists 😊

3. Data Hiding:

What is data hiding?

- ✓ Data hiding is nothing but hiding of the data.

Why should we hide?

- ✓ Based on requirement sometimes we need to hide the data
- ✓ If we hide the data, then outside class can't access our data directly.

How to hide the data?

- ✓ By using private modifier, we can implement data hiding.
- ✓ The main advantage of data hiding is we can achieve security.

Program Name: Without using private keyword
Demo4.scala

```
class SbiAccount
{
        val balance: Double = 500
}

class HdfcBank
{
        def bankBalance()
        {
                val s = new SbiAccount()
                println(s.balance)
        }
}


object Demo4
{
        def main(args: Array[String])
        {
                val h = new HdfcBank()
                h.bankBalance ()
        }
}
```

Compile: scalac Demo4.scala
Run: scala Demo4

Output

```
500.0
```

Program Name: Data hiding by using private keyword
Demo5.scala

```
class SbiAccount
{
        private val balance: Double = 500;
}

class HdfcBank
{
        def bankBalance()
        {
                val a = new SbiAccount()
                println(a.balance)
        }
}

object Demo5
{
        def main(args: Array[String])
        {
                val h = new HdfcBank()
                h.bankBalance ()
        }
}
```

Compile: scalac Demo5.scala
Run: scala Demo5

Output

```
error: value balance in class SbiAccount cannot be accessed in SbiAccount
```

4. Abstraction

Definition 1:

- ✓ Abstraction means hiding the unnecessary data from the user.

Definition 2:

- ✓ Technically speaking abstraction means
  - **H**iding internal implementation details

    &
  - **H**ighlight the set of services what are offering.

Example:

- ✓ In bank ATM application, its highlight the set of services,
  - withdraw
  - balance
  - mini statement
- ✓ In bank ATM application used to hide,
  - Internal implementation.
- ✓ The main advantage of abstraction is we can achieve security.

5. Encapsulation:

- ✓ Binding of the data and corresponding methods into a single unit is called "Encapsulation".
- ✓ Encapsulation = Data Hiding + Abstraction.
- ✓ If any scala class follows Data hiding & abstraction such type of class is called as an encapsulated class.

- ✓ Example: A class is best example for Encapsulation.
- ✓ The central concept of Encapsulation is hiding data behind methods.

Methods

- ✓ We can define a method by using def keyword
- ✓ The purpose of method is to perform operations in class.
- ✓ Terminology related to methods,
    - def keyword
    - method name
    - parenthesis
    - parameters (if required)
    - method body
    - return type (if required)
    - = symbol
- ✓ After creating the method then we need to call that method to do operation.

Make a note

- ✓ Method name along with its parameters is called method signature.

Types of methods

- ✓ Based on parameters methods are divided into two types,

1. Zero parameterised methods
2. Parameterized methods

Zero parameterized methods

- ✓ If method having no parameters, then those methods are called as zero parameterized method.

Program Name: Creating zero parameterised method and accessing by using object Demo6.scala

```
class Test
{
        def m()
        {
                println("Welcome to methods concept")
        }
}

object Demo6
{
        def main(args: Array[String])
        {
                val t = new Test()
                t.m()
        }
}
```

Compile: scalac Demo6.scala

Run: scala Demo6

Output:

```
Welcome to methods concept
```

Program Name: Creating zero parameterised method and accessing by using object
Demo7.scala

```
class Test
{
	def m()
	{
		var a=10

		if(a==10)
		{
			println("a value is: "+a)
		}

		else
		{
			println("a value is not 10")
		}
	}
}

object Demo7
{
	def main(args: Array[String])
	{
		val t = new Test()
		t.m()
	}
}
```

Compile
scalac Demo7.scala

Run
scala Demo7

Output

```
a value is: 10
```

Make a note

- ✓ If method having no parameters, then we can ignore parenthesis while calling method.

Program Name: If method having no parameters then parenthesis is options while calling
Demo8.scala

```
class Test
{
        def m()
        {
                println("Welcome to methods concept")
        }
}

object Demo8
{
        def main(args: Array[String])
        {
                val t = new Test()
                t.m
        }
}
```

Compile: scalac Demo8.scala
Run: scala Demo8

Output

```
Welcome to methods concept
```

Parameterized methods

- ✓ If method having parameters, then those methods called as parameterized methods.
- ✓ If method having parameters, then while calling those methods we need to pass values

Program Name: Creating parameterised method and accessing by using object
Demo9.scala

```
class Test
{
        def display(x: Int, y: Int)
        {
                println(x)
                println(y)
        }
}

object Demo9
{
        def main(args: Array[String])
        {
                val t = new Test()
                t.display(10, 20)
        }
}
```

Compile: scalac Demo9.scala
Run: scala Demo9

Output

```
10
20
```

Sometimes Method may not be having curly braces

- ✓ This is purely for simplicity.
- ✓ Whenever code of the method is small then we can ignore the braces.
- ✓ When the code of the method is bigger then, it's good to write within curly braces.

Program Name: Sometimes method may not be having curly braces
Demo10.scala

```
class Demo1
{
        def max(x:Int, y:Int): Int = if (x>y) x else y
}

object Demo10
{
        def main(args: Array[String])
        {
                val d = new Demo1()
                println(d.max(10, 20))
        }
}
```

Compile: scalac Demo10.scala
Run: scala Demo10

Output

```
20
```

return keyword

- ✓ return is a keyword.
- ✓ Writing a program only by using method is valid
- ✓ Writing a program method + return also valid

Syntax

```
class NameOfTheClass
{
        def methodName(): DataType=
        {
                return 100
        }
}
```

- ✓ If method having return statement,
    - We need to write a data type to method by using colon separator.
- ✓ After data type we need to write equals (=) symbol
- ✓ We can return any type of data type

Example 1

```
class Student
{
        def name(): String=
        {
                return "Nireekshan"
        }
}
```

Example 2

```
class Bank
{
        def balance(): Int=
        {
                return 100
        }
}
```

Program Name: Creating Bank class and method
Demo11.scala

```
class Bank
{
        def balance()
        {
                println("My balance is: ")
        }
}

object Demo11
{
        def main(args: Array[String])
        {
                val b = new Bank()
                b.balance()
        }
}
```

Compile: scalac Demo11.scala
Run: scala Demo11

Output

```
My balance is:
```

Program Name: using return type
Demo12.scala

```
class Bank
{
        def balance(): Int=
        {
                print("My balance is: ")
                return 100
        }
}

object Demo12
{
        def main(args: Array[String])
        {
                val b = new Bank()
                val bal = b. balance()
                print(bal)
        }
}
```

Compile: scalac Demo12.scala
Run: scala Demo12

Output

```
My balance is: 100
```

Make a note

- ✓ If method having return statement, then method calling we need to assign to a variable.
- ✓ This assigned variable holds the return value.

Why we need to assign Nireekshan?

- ✓ Good question.
- ✓ That assigned variable we can use further level in program
- ✓ Just observe below program

Program Name: using return type
Demo13.scala

```
class Bank
{
        def balance(): Int=
        {
                return 100
        }
}

object Demo13
{
        def main(args: Array[String])
        {
                val b = new Bank()

                val bal = b. balance()


                if(bal==0)
                {
                        println("Balance is zero ")
                }

                else if(bal<0)
                {
                        println("Balance is negative ")
                }

                else
                {
                        println("Balance is: "+bal)
                }
        }
}
```

Compile: scalac Demo13.scala
Run: scala Demo13

Output:

```
Balance is: 100
```

3. Constructors in scala

3.1 Purpose of constructor

- ✓ To initialize the instance variables.

3.2 When constructor will get execute?

- ✓ We no need to call constructor explicitly.
- ✓ Constructor executes automatically during object creation. (Demo15.scala)

3.3. How many times constructor will get execute?

- ✓ How many times we create objects that many times constructor will get execute.
- ✓ If we create 10objects, then 10times it executes.

How to define constructor?

- ✓ In scala, the syntax of first constructor used to define along with class only.

Program Name: Constructor
Demo14.scala

```
class Student()
{
        println("Constructor")
}

object Demo14
{
        def main(args: Array[String])
        {
                println("Welcome to main method ")
        }
}
```

Compile: scalac Demo14.scala
Run: scala Demo14

Output

```
Welcome to main method
```

Program Name: Constructor
Demo15.scala

```
class Student()
{
        println("Constructor")
}


object Demo15
{
        def main(args: Array[String])
        {
                val s=new Student()
        }
}
```

Compile: scalac Demo15.scala
Run: scala Demo15

Output

```
Constructor
```

Program Name: Constructor
Demo16.scala

```
class Student()
{
        println("Constructor")
}

object Demo16
{
        def main(args: Array[String])
        {
                val s1 = new Student()
                val s2 = new Student()
        }
}
```

Compile: scalac Demo16.scala
Run: scala Demo16

Output

```
Constructor
Constructor
```

Make a note

- ✓ Developer no needs to call explicitly.
- ✓ During object creation constructor executes automatically.

Make a note

- ✓ Developer need to call methods explicitly, but not constructor.

3.2 Types of constructor

- ✓ Primary constructor
  - without parameters
  - with parameters
- ✓ Auxiliary constructor

3.1.1 Primary constructor without parameters

- ✓ In scala, the syntax of first constructor used to define along with class only.
- ✓ It helps to optimize code.
- ✓ If constructor having no parameters, the it is called as zero parameterized constructor.

Program Name: Constructor
Demo17.scala

```
class Student()
{
        println("Constructor")
}


object Demo17
{
        def main(args: Array[String])
        {
                val s=new Student()
        }
}
```

Compile: scalac Demo17.scala
Run: scala Demo17

Output

```
Constructor
```

Make a note:

- ✓ In scala, if you don't specify primary constructor then compiler creates a constructor automatically. (practically you can check by using scalap command)
- ✓ Based on requirement a class can contains any number of constructors.

3.1.2 Primary constructor with parameters

- ✓ If constructor having parameters, then we can called as parameterised constructor.
- ✓ If constructor having parameters, then during object creation we need to pass values to that parameterised constructor.

Program Name: Constructor with parameters
Demo18.scala

```
class Employee(name: String, age: Int)
{
        println("Name is: " +name)
        println("Age is sweet: " +age)
}


object Demo18
{
        def main(args: Array[String])
        {
                var e = new Employee("Nireekshan", 16);
        }
}
```

Compile: scalac Demo18.scala
Run: scala Demo18

Output

```
Name is: Nireekshan
Age is sweet: 16
```

Program Name: Constructor with parameters
Demo19.scala

```
class Employee(name: String, age: Int)
{
        def showDetails()
        {
                println("Name is: " +name)
                println("Age is sweet: " +age)
        }
}


object Demo19
{
        def main(args: Array[String])
        {
                var e = new Employee("Nireekshan", 16);
                e.showDetails()
        }
}
```

Compile: scalac Demo19.scala
Run: scala Demo19

Output

```
Name is: Nireekshan
Age is sweet: 16
```

2 Auxiliary Constructor

- ✓ Auxiliary constructor also called as Secondary constructor.
- ✓ Based on requirement we can create more than one constructor in a class
- ✓ By using this, we can create Auxiliary constructors.

Rules to define Auxiliary constructor

- ✓ We can create Auxiliary constructor by using this
- ✓ We must call primary constructor from auxiliary constructor.
- ✓ By using this keyword, we can call the constructor from one to another.
- ✓ Whenever we are calling another constructor then the calling code should be first piece of code.

Program Name: Auxiliary Constructor with parameters
Demo20.scala

```
class Employee(id: Int, name: String)
{
        var age: Int = 0

        def this(id: Int, name: String, age: Int)
        {
                this(id, name)    // Calling primary constructor

                this.age = age
        }


        def showDetails()
        {
                println("id is: "+id)
                println("Name is: "+name)
                println("Age is sweet: "+age)
        }
}


object Demo20
{
        def main(args: Array[String])
        {
                var emp = new Employee(101,"Nireekshan",16);
                emp.showDetails()
        }
}
```

Compile: scalac Demo20.scala
Run: scala Demo20

Output

```
id is:101
Name is: Nireekshan
Age is sweet: 16
```

Make a note

- ✓ If instance variable name and parameter names are same, then to define instance variables we need to use this keyword on variables (Please observe above example)

Difference between constructor and method

Method	Constructor
✓ Purpose: Methods are used to perform operations	✓ Purpose: Constructors are used to initialize the instance variables.
✓ Name: Method name can be any name.	✓ Name: If auxiliary constructor then name should be this()
✓ Access: Methods we should call explicitly to execute	✓ Access: Constructor automatically executed at the time of object creation.

Types of methods

- ✓ Based on implementations methods are divided into two types,
  - Instance methods
  - Singleton methods
- ✓ This concept we will discuss in upcoming chapter which is normal class and singleton class or object

10. OOPS – Part – 2 - Inheritance

What is inheritance?

- ✓ Creating new classes from already existing classes is called as inheritance.
- ✓ The existing class is called a super class or base class or parent class.
- ✓ The new class is called as sub class or derived class or child class.
- ✓ Inheritance allows sub classes to inherit the variables, methods and constructors of their super class.
    - ✓ Except the private variables and methods.
- ✓ One class can extend only one class at a time.
- ✓ One class cannot extend more than one class, because scala does not support multiple inheritance.

Make a note

- ✓ Without Inheritance we can't write even a simple Scala program also.
- ✓ Our First Hello World program is a child class to **Any** class in scala.
- ✓ Any class is pre-defined super class for every class in scala.
    - Any super class is available in scala package.

How to implement inheritance?

- ✓ By using extends keyword we can implement the inheritance.

Advantages of Inheritance:

- ✓ Application development time is very less.
- ✓ Redundancy (repetition) of the code is reducing.

Tip

- ➢ Frankly tell me Boss, did you understand inheritance or not.
- ➢ If not, then please read it one more time after having cup of coffee.

Program Name: Creating two class and applying inheritance concept
Demo1.scala

```
class One
{
	def m1()
	{
		println("m1 method from parent class")
	}
}

class Two extends One
{
	def m2()
	{
		println("m2 method from child class")
	}
}


object Demo1
{
	def main(args: Array[String])
	{
		val t = new Two()

		t.m1()
		t.m2()
	}
}
```

Compile: scalac Demo1.scala
Run: scala Demo1

Output

```
m1 method from parent class
m2 method from child class
```

Types of Inheritance:

1. Single Inheritance
2. Multilevel inheritance
3. Multiple inheritance

1. Single Inheritance:

✓ Creating a sub class from a single super class is called single inheritance.

Program Name: Creating two class and applying inheritance concept
Demo2.scala

```
class Parent
{
        def properties()
        {
                println("money + land + gold")
        }
}

class Child extends Parent
{
        def study()
        {
                println("Studies done and waiting for job to get marriage")
                println("Requesting please do prayer for my job")
        }
}


object Demo2
{
        def main(args: Array[String])
        {
                val c = new Child()

                c.properties()
                c.study()
        }
}
```

Compile: scalac Demo2.scala
Run: scala Demo2

Output

```
money + land + gold
Studies done and waiting for job to get marriage
Requesting please do prayer for my job
```

Program Name: Creating two class and applying inheritance concept
Demo3.scala

```
class Parent
{
	var a: Int = 10
	var b: Int = 20

	def m1()
	{
		println("a value from parent: "+a)
		println("b value from parent: "+b)
	}
}


class Child extends Parent
{
	var d: Int = 30
	var e: Int = 40

	def m2()
	{
		println("d value from child: "+d)
		println("e value from child: "+e)
	}
}


object Demo3
{
	def main(args: Array[String])
	{
		val c = new Child()

		c.m1()
		c.m2()
	}
}
```

Compile: scalac Demo3.scala
Run: scala Demo3

Output

```
a value from parent: 10
b value from parent: 20
d value from child: 30
e value from child: 40
```

Make a note

- ✓ Private data members not involve in Inheritance

Program Name: Creating two class and applying inheritance concept
Demo4.scala

```
class Parent
{
        private def m1()
        {
                println("private method m1 from parent class")
        }
}

class Child extends Parent
{
        def m2()
        {
                println("m2 method from child class")
        }
}

object Demo4
{
        def main(args: Array[String])
        {
                val c = new Child()

                c.m1()
                c.m2()
        }
}
```

Compile: scalac Demo4.scala
Run: scala Demo4

Output

```
error: value m1 is not a member of Child
```

2. Multi-level Inheritance:

- ✓ A class is derived from another derived class is called multi-level inheritance

Program Name: Creating two class and applying inheritance concept
Demo5.scala

```
class GrandFather
{
        def gfProperties()
        {
                println("only land from grandfather")
        }
}

class Father extends GrandFather
{
        def fProperties()
        {
                println("money + land + gold from father")
        }
}

class Child extends Father
{
        def study()
        {
                println("Studies done and waiting for job to get marriage")
                println("Requesting please do prayer for my job")
        }
}

object Demo5
{
        def main(args: Array[String])
        {
                val c = new Child()

                c.gfProperties()
                c.fProperties()
                c.study()
        }
}
```

Compile: scalac Demo5.scala
Run: scala Demo5

Output

```
only land from grandfather
money + land + gold
Studies done and waiting for job to get marriage
Requesting please do prayer for my job
```

Program Name: Creating two class and applying inheritance concept
Demo6.scala

```
class A
{
	var p: Int = 10
	var q: Int = 20;

	def m1()
	{
		println("p value : "+p)
		println("q value : "+q)
	}
}

class B extends A
{
	var r: Int = 30
	var s: Int = 40

	def m2()
	{
		println("r value : "+r)
		println("s value : "+s)
	}
}

class C extends B
{
	var t: Int = 50
	var u: Int = 60

	def m3()
	{
		println("t value : "+t)
		println("u value : "+u)
	}
}

object Demo6
{
	def main(args: Array[String])
	{
		val d = new C()

		d.m1()
		d.m2()
		d.m3()
	}
}
```

Compile: scalac Demo6.scala
Run: scala Demo6

Output

```
p value : 10
q value : 20
```

```
r value : 30
s value : 40
t value : 50
u value : 60
```

3. Multiple Inheritance:

- ✓ Creating a sub class from multiple super classes is called multiple inheritance.
- ✓ But java and Scala does not support multiple inheritance.

Why multiple inheritance is not supporting?

- ✓ There may be a chance of, two super classes may be having same variables or methods names, then the child will get ambiguity while accessing.

Program Name: Trying to create a class from two parent classes
Demo7.scala

```
class A
{
        var i: Int = 10
}

class B
{
        var i: Int = 10
}

class C extends A, B
{
        var k=20
}

class Demo7
{
        def main(args: Array[String])
        {
                val c = new C()
                print(c.i)
        }
}
```

Compile: scalac Demo7.scala
Run: scala Demo7

Output

```
error: ';' expected but ',' found.
class C extends A, B
                 ^
one error found
```

11. OOPS – Part – 3 - Polymorphism

What is Polymorphism?

- ✓ The process of representing "one form in many forms".
- ✓ Poly means many.
- ✓ Morphs means forms.
- ✓ Polymorphism means 'Many Forms'.

What is polymorphism?

- ✓ The ability to exist in different forms is called "Polymorphism".
- ✓ In scala an object or a method can exist in different forms, thus performing various tasks depending on the context.

Make a note

- ✓ This point is only for Java guys, remaining guys please get relax.
- ✓ In scala there is no static polymorphism, because no static keyword in scala.
- ✓ In scala only one polymorphism that is dynamic polymorphism.

Method parameters

- ✓ We can create a method which having parameters as well.

Program Name: Method can contain parameters
Demo1.scala

```
class Sum
{
        def add(a: Int, b: Int)
        {
                println("Sum of two numbers: "+(a+b))
        }

}


object Demo1
{
        def main(args: Array[String])
        {
                val s=new Sum()

                s.add(10,20)
        }
}
```

Compile: scalac Demo1.scala
Run: scala Demo1

Output

```
Sum of two numbers: 30
```

Make a note

- ✓ In above program **add** is a method name **a** and **b** are called as parameters

Dynamic Polymorphism

- ✓ This is also called run time polymorphism.
- ✓ The polymorphism which is exhibited at runtime is called dynamic binding.
- ✓ The JVM only knows which one (variable or method) supposed to be execute at run time.

Program Name: Dynamic polymorphism
Demo2.scala

```
class Sum
{
        def add(a: Int, b: Int)
        {
                println("Sum of two numbers: "+(a+b))
        }

        def add(a: Int, b: Int, c: Int)
        {
                println("Sum of three numbers: "+(a+b+c))
        }
}

object Demo2
{
        def main(args: Array[String])
        {
                val s=new Sum()

                s.add(10,20)
                s.add(10,20,30)
        }
}
```

Compile: scalac Demo2.scala
Run: scala Demo2

Output

```
Sum of two numbers: 30
Sum of three numbers: 60
```

Examples for dynamic Polymorphism

- ✓ Method overloading
- ✓ Method overriding

Method Overloading:

- ✓ In a class writing two or more methods with the same name but with difference parameters is called method overloading.

Program Name: Method overloading
Demo3.scala

```
class Sum
{
        def add(a: Int, b: Int)
        {
                println("Sum of two numbers: "+(a+b))
        }

        def add(a: Int, b: Int, c: Int)
        {
                println("Sum of three numbers: "+(a+b+c))
        }
}

object Demo3
{
        def main(args: Array[String])
        {
                val s=new Sum()

                s.add(10,20)
                s.add(10,20,30)
        }
}
```

Compile: scalac Demo3.scala
Run: scala Demo3

Output

```
Sum of two numbers: 30
Sum of three numbers: 60
```

Cases in overloading:

- ✓ In method overloading three cases are available

1. Difference in number of parameters
2. Difference in type of parameters
3. Difference in order of parameters

Case 1: Difference in number of parameters

- ✓ In overloading we can define two methods having same name with different number of parameters

Program Name: Case 1: Difference in number of parameters
Demo4.scala

```
class Addition
{
	def add(a: Int, b: Int)
	{
		println(a + b)
	}


	def add(a: Int, b: Int, c: Int)
	{
		println(a + b + c)
	}
}

object Demo4
{
	def main (args: Array[String])
	{
		val a = new Addition()

		a.add(40,40)
		a.add(20,20,20)
	}
}
```

Compile: scalac Demo4.scala
Run: scala Demo4

Output

```
80
60
```

Case 2: Difference in type of parameters

- ✓ In overloading we can define two methods having same name with different type of parameters

Program Name: Case 2: Difference in type of parameters
Demo5.scala

```
class Addition
{
	def add(a: Int, b: Int)
	{
		println(a + b)
	}


	def add(a: Double, b: Double)
	{
		println(a + b)
	}
}

object Demo5
{
	def main(args: Array[String])
	{
		val a = new Addition()

		a.add(40,40)
		a.add(20.1,20.3)
	}
}
```

Compile: scalac Demo5.scala
Run: scala Demo5

Output

```
80
40.400
```

Case 3: Difference in order of parameters

- ✓ In overloading we can define two methods having same name with different order of parameters

Program Name: Case 3: Difference in order of parameters
Demo6.scala

```
class Addition
{
        def add (a: Int, b: Double)
        {
                println(a + b)
        }


        def add (a: Double, b: Int)
        {
                println(a + b)
        }
}

object Demo6
{
        def main (args: Array[String])
        {
                val a = new Addition()

                a.add(40,40.12)
                a.add(20.56,20)
        }
}
```

Compile: scalac Demo6.scala
Run: scala Demo6

Output

```
80.12
40.56
```

Method overriding

How to implement method overriding?

- ✓ We can implement method overriding by using override keyword

What is method overriding?

- ✓ Writing a method in super class and sub class which having same name and same parameters.

Program Name: Creating two class and applying inheritance concept
Demo7.scala

```
class Parent
{
	def m1()
	{
		println("Parent - m1")
	}
}

class Child extends Parent
{
	override def m1()
	{
		println("Child - m1")
	}
}

object Demo7
{
	def main(args: Array[String])
	{
		val c = new Child()
		c.m1()
	}
}
```

Compile: scalac Demo7.scala
Run: scala Demo7

Output

Child – m1

When should we go for overriding?

- ✓ If child class won't like parent class method implementation, then happily child class can override parent class method.

Program Name: Creating two class and applying inheritance concept
Demo8.scala

```
class Parent
{
        def properties()
        {
                println("money + land + gold")
        }

        def marriage()
        {
                println("Father decided Child marriage with uncle daughter: Her
                name is Subbalaxmi")
        }
}

class Child extends Parent
{
        def study()
        {
                println("Studies done and got job")
                println("Thank you all for your prayers")
        }

}

object Demo8
{
        def main(args: Array[String])
        {
                val c = new Child()

                c.properties()
                c.study()
                c.marriage()
        }
}
```

Compile: scalac Demo8.scala
Run: scala Demo8

Output

```
money + land + gold
Studies done and got job
Thank you all for your prayers
Father decided Child marriage with uncle daughter: Her name is Subbalaxmi
```

Program Name: Creating two class and applying inheritance concept
Demo9.scala

```
class Parent
{
	def properties()
	{
		println("money + land + gold")
	}

	def marriage()
	{
		println("Father decided Child marriage with uncles daughter: Her
		name is Subbalaxmi")
	}
}

class Child extends Parent
{
	def study()
	{
		println("Studies done and got job")
		println("Thank you all for your prayers")
	}

	override def marriage()
	{
		println("Child wont like father decision about regarding
		marriage, so planning to marry Anushka in Banglore")
	}
}

object Demo9
{
	def main(args: Array[String])
	{
		val c = new Child()

		c.properties()
		c.study()
		c.marriage()
	}
}
```

Compile: scalac Demo9.scala
Run: scala Demo9

Output

```
money + land + gold
Studies done and got job
Thank you all for your prayers
Child wont like father decision about regarding marriage, so planning to marry
Anushka in Banglore
```

Program Name: Creating two class and applying inheritance concept
Demo10.scala

```
class Commercial
{
	def electricity()
	{
		println ("Welcome to Electricity");
	}

	def calculateBill(units: Int)
	{
		println ("Commercial Bill amount: "+units*5.00);
	}
}

class Domestic extends Commercial
{
	override def calculateBill(units: Int)
	{
		println("Domestic Bill amount: "+units*2.00);
	}
}

object Demo10
{
	def main (args: Array[String])
	{
		val c = new Commercial()
		c.electricity
		c.calculateBill(100)

		val d=new Domestic()
		d.electricity
		d.calculateBill(100)
	}
}
```

Compile
scalac Demo10.scala

Run
scala Demo10

Output

```
Welcome to Electricity
Commercial Bill amount: 500.0
Welcome to Electricity
Domestic Bill amount: 200.0
```

Difference between Method overloading and Method overriding

Overloading	Overriding
✓ Writing two or more methods with the same name but different parameters is called method overloading.	✓ Writing two or more methods with the same name with same parameters is called method overriding.
✓ No keyword is required.	✓ By using override keyword.
✓ Method overloading is done in the same class.	✓ Method overriding is done in super and sub classes, so here inheritance involves.
✓ In method overloading method return type can be same or different	✓ In method overriding method return type should be same.

final keyword

- ✓ In scala final keyword we can apply on two concepts,

    1. method
    2. class

- ✓ So, in scala,

    1. A method can be final
    2. A class can be final

1. final method

- ✓ In super class, if we declare a method as a final then, it is not possible to override this method in child class.
- ✓ So, final methods cannot be overridden

Program Name: Trying to override final method
Demo11.scala

```
class Parent
{
        def properties()
        {
                println("money + land + gold")
        }


        final def marriage()
        {
                println("Father decided Child marriage with uncles daughter: Her
                name is Subbalaxmi")
        }
}

class Child extends Parent
{
        def study()
        {
                println("Studies done and got job")
                println("Thank you all for your prayers")
        }

        override def marriage()
        {
                println("Child wont like father decision about regarding
                marriage, so planning to marry Anushka in Banglore")
        }
}

object Demo11
{
        def main(args: Array[String])
        {
                val c = new Child()

                c.properties()
                c.study()
                c.marriage()
        }
}
```

Compile: scalac Demo11.scala
Run: scala Demo11

Output

```
overriding method marriage in class Parent of type ()Unit;
method marriage cannot override final member
override def marriage()
                    ^
```

2. final class

- ✓ If we declare a class as a final, then it is not possible to inherit this class.
- ✓ Final classes cannot be inherited

Program Name: Trying to inherit final class
Demo12.scala

```
final class Parent
{
        def m1()
        {
                println("m1 method from parent class")
        }
}

class Child extends Parent
{
        def m2()
        {
                println("m2 method from child class")
        }
}

object Demo12
{
        def main(args: Array[String])
        {
                val c = new Child()

                c.m1()
                c.m2()
        }
}
```

Compile: scalac Demo12.scala
Run: scala Demo12

Output

```
error: illegal inheritance from final class Parent
class Child extends Parent
                    ^
```

Summary of the story

- ✓ final methods cannot be overridden.
- ✓ final classes cannot be inherited.

Smart question: If we are using final keyword then, Are we missing OOPs features?

- ✓ Yes Boss☹, if you are using final keyword then we are missing inheritance and overriding concepts.
- ✓ If it is really required, then only use final keyword otherwise enjoy oops features cheers.

12. OOPS – Part – 4 - abstract class, trait

abstract keyword

- ✓ abstract is a keyword in scala.
- ✓ We can apply abstract keyword on three concepts,

1. class
2. method
3. variable

- ✓ So, in scala,

1. A class can be abstract
2. A method can be abstract
3. A variable can be abstract

Just recall once scala method

- ✓ As we discussed method have two parts,

1. method name and parameters (if exists)
2. method body

```
class Bank
{
        def balance()
        {
                println ("This is body of the method")
        }
}
```

There are two types of methods in-terms of implementation

1. Implemented methods.
2. Un-implemented method.

1. Implemented method

- ✓ A method which have a method name and method body then that method is called as implemented method.
- ✓ Also called as concrete method or non-abstract method

```
class Bank
{
        def balance()
        {
                println ("This is body of the method")
        }
}
```

2. Un-implemented method

- ✓ A method which has only method name and no method body then that method is called as un-implemented method.
- ✓ Also called as non-concrete or abstract method.

```
abstract class Bank
{
        def interest()
}
```

- ✓ In above code, interest() method having no method body.
- ✓ So, this method is called as abstract method.

abstract method

- ✓ abstract class and trait can contain abstract methods.
- ✓ abstract method will not have method body.
- ✓ abstract method will be implemented in its sub class of abstract class.
- ✓ Explicitly we no need to give abstract keyword for abstract method.
- ✓ If any method having no method body means automatically that will become an abstract method.

Syntax

```
abstract class NameOfTheClass
{
        def nameOfTheMethod()
}
```

Example 1

```
abstract class Bank
{
        def interest()
        def offers()
}
```

Make a note

- ✓ If any class having abstract method, then that class should be declared as an abstract class.

abstract class

- ✓ We can create abstract class by using abstract keyword.
- ✓ A class which is declared as abstract is known as abstract class.
- ✓ abstract class can contain,
    - constructors
    - abstract variables
    - non-abstract variables
    - abstract methods
    - non-abstract methods
    - sub class
- ✓ abstract methods should be implemented in sub class of abstract class. (Demo48.scala)
- ✓ If sub class didn't provide implementation of abstract method, then we need to declare that sub class as abstract class.(Demo49.scala)
- ✓ If any class inheriting this sub class, then that sub class should provide the implementation for abstract methods. (Demo49.scala)
- ✓ *object creation is not possible for abstract class*. (Demo50.scala)

Reminder

- ✓ If any class having abstract method, then that class should be declared as an abstract class.

Syntax

```
abstract class NameOfTheClass
{
        Mainly it can contain,

        1. abstract methods
        2. non-abstract methods
}
```

Program Name: Abstract class and child class giving implementation for abstract methods
Demo1.scala

```
abstract class Bank
{
	def balanceCheck()
	{
		println("Balance checking implementation ")
	}

	def transfer()
	{
		println("transfer implementation ")
	}

	def interest()
}

class Sbi extends Bank
{
	def interest()
	{
		println("Sbi bank interest is 10 rupees")
	}
}

object Demo1
{
	def main(args: Array[String])
	{
		val s = new Sbi()

		s.balanceCheck()
		s.transfer()
		s.interest()
	}
}
```

Compile: scalac Demo1.scala
Run: scala Demo1

Output

```
Balance checking implementation
transfer implementation
Sbi bank interest is 10 rupees
```

Program Name: Abstract class and child class giving implementation for abstract methods
Demo2.scala

```
abstract class Bank
{

	def balanceCheck()
	{
		println("Balance checking implementation ")
	}

	def transfer()
	{
		println("transfer implementation ")
	}

	def interest()
}

abstract class Sbi extends Bank
{
	def offers()
	{
		println("Sbi bank having good offers")
	}
}

class Sbi1 extends Sbi
{
	def interest()
	{
		println("Sbi bank interest is 10 rupees")
	}
}

object Demo2
{
	def main(args: Array[String])
	{
		val s = new Sbi1()

		s.balanceCheck()
		s.transfer
		s.offers()
		s.interest()
	}
}
```

Compile
scalac Demo2.scala

Run
scala Demo2

Output

```
Balance checking implementation
transfer implementation
Sbi bank having good offers
```

```
Sbi bank interest is 10 rupees
```

Program Name: object creation is not possible for abstract class
Demo3.scala

```
abstract class Bank
{

        def balanceCheck()
        {
                println("Balance checking implementation ")
        }

        def transfer()
        {
                println("transfer implementation ")
        }

        def interest()
}

object Demo3
{
        def main(args: Array[String])
        {
                val s = new Bank()

        }
}
```

Compile: scalac Demo3.scala
Run: scala Demo3

Output

```
error: class Bank is abstract; cannot be instantiated
val s = new Bank()
                ^
```

trait

trait

- ✓ trait is a keyword in scala

- ✓ This point is for Java guys:
  - By using trait keyword, we can create trait just like an interface in java

What is trait?

- ✓ A trait is just like an interface in java.
- ✓ We can create trait by using trait keyword.
- ✓ trait can contain,
  - abstract variables
  - non-abstract variables
  - abstract methods
  - default methods (non-abstract methods)
  - sub class

- ✓ abstract methods will be implemented in sub class of trait. (Demo56.scala)
- ✓ If sub class didn't provide implementation of abstract method, then we need to declare that sub class as abstract class. (Demo57.scala)
- ✓ If any class inheriting this sub class, then that sub class should provide the implementation for abstract methods. (Demo57.scala)
- ✓ object creation is not possible for trait(Demo58.scala)

Points to remember

- ✓ One class can extend any number of traits by using with keyword. (Demo.scala)
- ✓ one trait can extend multiple traits.(Demo.scala)
- ✓ Trait cannot have constructors.
- ✓ Trait is like an interface in Java.

Make a note

- ✓ In trait non-abstract methods are default methods.
- ✓ These default methods are by-default available to the child classes of traits.

Syntax

```
trait NameOfTheTrait
{
        Mainly it can contain,

        1. abstract methods
        2. default methods(non-abstract methods)
}
```

Program Name: Creating trait and child class for trait
Demo1.scala

```
trait Bank
{
	def info()
	{
		println("This is bank application")
	}

	def interest()
}

class AndhraBank extends Bank
{
	def interest()
	{
		println("Interest is 10 rupees")
	}
}

object Demo1
{
	def main (args: Array[String])
	{
		val a = new AndhraBank()

		a.info()
		a.interest()
	}
}
```

Compile: scalac Demo1.scala
Run: scala Demo1

Output

```
This is bank application
Interest is 10 rupees
```

Program Name: Creating trait and child classes for trait
Demo2.scala

```
trait Bank
{
	def info()
	{
		println("This is bank application")
	}

	def interest()
}

abstract class TelanganaBank extends Bank
{
	def offers()
	{
		println("Giving silver coin for new customers")
	}
}

class TelanganaBankSub1 extends TelanganaBank
{
	def interest()
	{
		println("Interest is 5 rupees")
	}
}

object Demo2
{
	def main(args: Array[String])
	{
		val d = new TelanganaBankSub1()

		d.info()
		d.offers()
		d.interest()
	}
}
```

Compile: scalac Demo2.scala
Run: scala Demo2

Output

```
This is bank application
Giving silver coin for new customers
Interest is 5 rupees
```

Program Name: Object creation is not possible for trait
Demo3.scala

```
trait A
{
        def m()
        def n()
}

object Demo3
{
        def main(args: Array[String])
        {
                val d = new A()
        }
}
```

Compile: scalac Demo3.scala
Run: scala Demo3

Output

```
error: trait A is abstract; cannot be instantiated
val d = new A()
                ^
```

- ✓ A single class can extend multiple traits

Program Name: Class is inheriting two child classes
Demo4.scala

```
trait Amazon
{
	def amazonShopping()

	def amazonInfo()
	{
		println("Welcome to Amazon shopping")
	}
}

trait FlipKart
{
	def flipKartShopping()

	def flipKartInfo()
	{
		println("Welcome to FlipKart shopping")
	}
}

class Customer extends Amazon with FlipKart
{
	def amazonShopping()
	{
		println("Bought Ponds powder dabba from amazon")
	}


	def flipKartShopping()
	{
		println("Bought hTC mobile from flipKart")
	}

}

object Demo4
{
	def main(args: Array[String])
	{
		val c = new Customer()

		c.amazonInfo()
		c.amazonShopping()

		c.flipKartInfo()
		c.flipKartShopping()

	}
}
```

Compile scalac Demo4.scala
Run scalac Demo4

Output

```
Welcome to Amazon shopping
Bought Ponds powder dabba from amazon
Welcome to FlipKart shopping
Bought hTC mobile from flipKart
```

Hey Nireekshan, can you explain, when should we go for class, abstract class and trait?

class

- ✓ If we know complete implementation about the requirements, then we should go for class.
- ✓ A class having complete implementation.

abstract class

- ✓ If we know partial implementation about the requirements, then we should go for abstract class.
- ✓ Abstract class can contain implemented and un-implemented methods as well.

trait

- ✓ If we don't know complete implementation about the requirements, then we should go for trait.

13. OOPS – Part 5 - Normal, Singleton object and Companion object etc

Normal class

- ✓ Normal class we can create by using class keyword
- ✓ Inside normal class we can define instance variables and instance methods.

```
class NameOfTheClass
{
        // Instance variable
        // Instance method
}
```

Example

```
class NameOfTheClass
{
        var id = 101
        var name = "Nireekshan"

        def display()
        {
                println("Id is: "+id)
                println("Name is: "+name)
        }
}
```

- ✓ In above program *id* and *name* are instance variable
- ✓ display() method is an instance method
- ✓ Instance methods will use instance variables to perform operations or action.

Singleton object

- ✓ In Scala static keyword is not available, instead of static keyword we need to use singleton object to fulfil the requirement.
- ✓ Singleton object we can create by using object keyword
- ✓ Inside singleton object we can define singleton variables and singleton methods.

```
object NameOfTheSingleTonObject
{
        // singleton variable
        // singleton methods
}
```

What is the purpose of singleton object?

- ✓ Let us understand below example

Program Name: Instance variables
Demo1.scala

```
class Student (id: Int, name: String, collegeName: String)
{
        def showDetails()
        {
                println(id)
                println(name)
                println(collegeName)
        }
}

object Demo1
{
        def main(args: Array[String])
        {
                val s1 = new Student(1, "Arjun", "DVS college")
                val s2 = new Student(2, "Prasad", "DVS college")
                val s3 = new Student(3, "Nireekshan", "DVS college")

                println("First Student information")
                s1.showDetails()

                println("Second Student information")
                s2.showDetails()

                println("Third Student information")
                s3.showDetails()

        }
}
```

Compile
scalac Demo1.scala

Run
scala Demo1

Output

```
First Student information
1
Arjun
DVS college

Second Student information
2
Prasad
DVS college

Third Student information
3
Nireekshan
DVS college
```

What is instance variable?

- ✓ If value of the variable is changing from object to object such type of variable is called as instance variables.

What is singleton variable?

- ✓ If value of the variable is not changing from object to object such type of variable is called as singleton variables.
- ✓ Here, for singleton variables memory will be allocated only once and that variable we can reuse in everywhere.

Program explanation

- ✓ Above program id and name is changing from object to object.
- ✓ But college name is not changing from object to object, so this type of variable we should not declare at singleton level.
- ✓ So, to create singleton class we need to use object keyword

How to access singleton variables?

- ✓ We should access singleton variables and methods directly by using singleton object name

Program Name: Creating singleton object
Demo2.scala

```
class Student (id: Int, name: String, collegeName: String)
{
        def showDetails()
        {
                println (id)
                println (name)
                println (collegeName)
        }
}

object College
{
        val colName: String = "DVS college"
}

object Demo2
{
        def main(args: Array[String])
        {
                val s1 = new Student(1, "Arjun", College.colName)
                val s2 = new Student(2, "Ramesh", College.colName)
                val s3 = new Student(3, "Nireekshan", College.colName)

                println("First Student information")
                s1.showDetails()

                println("Second Student information")
                s2.showDetails()

                println("Third Student information")
                s3.showDetails()
        }
}
```

Compile
```
scalac Demo2.scala
```
Run
```
scala Demo2
```

Output
```
First Student information
1
Arjun
DVS college

Second Student information
2
Prasad
DVS college

Third Student information
3
Nireekshan
DVS college
```

Standalone class

- ✓ Standalone class we can create by using object keyword.
- ✓ A class which can contain main method is called as Standalone class

```
object NameOfTheStandAloneClass
{
        // main method
}
```

Examples

- ✓ Till we have seen many standalone classes which having main method

Scala Companion Object

- ✓ In Scala program, syntactically it is valid if we are declaring a normal class name and singleton class name as the same name.
- ✓ If we are giving normal class name and singleton class as same, then such type of classes is called as companion object.
- ✓ The companion object is useful for implementing helper methods and factory.

Advantage

- ✓ We can use companion object to create instances for a specific class without using new keyword.

Define a normal class

```
class Animal(name: String)
{
        def display()
        {
                println("Animal name is:"+name)
        }
}
```

Define companion object for a Animal class

Rules to follow:

- ✓ We can define companion object by using object keyword.
- ✓ Name of companion object and class name should be same.
- ✓ These two should be in same source file.

Companion object responsible

- ✓ Companion object should define an apply() method.
- ✓ Internally this method will be creating object for corresponding class.

Define a companion object

```
object Animal
{
        def apply(name: String): Animal =
        {
                new Animal(name)
        }
}
```

Creating object to Animal class

- ✓ Now happily we can create object for Animal class without using new keyword.

```
val d = Animal("Dog")
val c = Animal("Cat")

d.display()
c.display()
```

Program Name: Creating companion object
Demo3.scala

```
class Animal(name: String)
{
        def display()
        {
                println("Animal name is: "+name)
        }
}

object Animal
{
        def apply(name: String): Animal =
        {
                new Animal(name)
        }
}

object Demo3
{
        def main(args: Array[String])
        {
                val d = Animal("Dog ")
                val c = Animal("Cat")

                d.display()
                c.display()
        }
}
```

Compile: scalac Demo3.scala
Run: scala Demo3

Output

```
Animal name is: Dog
Animal name is: Cat
```

case class

- ✓ A class which is declared with case keyword is called as case class.

Why case class?

- ✓ It's just like normal class but internally it creates companion object automatically
- ✓ By default case classes will get few methods automatically,
    - apply()
    - toString()
    - hashCode()
    - equals()
- ✓ This point if for java guys, scala case classes will helpful to reduce boiler plate code.

Why above methods are required?

- ✓ After creating objects for a class, sometimes based on requirement its required to compare the objects related stuff.
- ✓ These comparisons will be done by above methods.
- ✓ In Java programming a java developer should write these methods explicitly in their programs.
- ✓ But in scala these methods are by default available for case classes.

Case class Advantages

- ✓ By default, hashCode, equals, toString methods are available.
- ✓ By default, classes are immutable.
- ✓ new keyword is not required to create object.

Difference between case classes and normal classes

- ✓ When you are comparing two normal classes' objects with **==** operator then it will compare the addresses of those two objects.
- ✓ When you are comparing two case classes' objects with **==** operator then it will compare the values of the objects.

Program Name: Creating normal class and comparing two objects
Demo4.scala

```
class Staff(name: String, age: Int)

object Demo4
{
        def main(args: Array[String])
        {
                val s1 = new Staff("David", 45)
                val s2 = new Staff("David",45)

                println(s1 == s2)       //      false
        }
}
```

Compile: scalac Demo4.scala
Run: scala Demo4

Output:

```
false
```

Program Name: Creating a case class comparing two objects
Demo5.scala

```
case class Staff(name: String, age: Int)

object Demo5
{
        def main(args: Array[String])
        {
                val s1 = Staff("David", 45)
                val s2 = Staff("David",45)

                println(s1 == s2)
        }
}
```

Compile: scalac Demo5.scala
Run: scala Demo5

Output:

```
true
```

14. Scala functional programming

General example why function required?

- ✓ When you go for walk in the early morning,

    1) Get up from the bed,
    2) Do fresh up,
    3) Tie the shoe,
    4) Pick the smooth towel,
    5) Start the walk.

- ✓ Think of this sequence of steps to do morning walk.
- ✓ Now when my dad calls for morning walk means, he doesn't want to explain all these steps each time.
- ✓ Whenever dad says, "Get ready for morning walk", means he is making a function call.
- ✓ Morning walk' is an abstraction for all the many steps involved.

When should we go for function?

Reason 1:

- ✓ While writing coding logics it's good to keep those coding statements in one separate block, because whenever required then we can call that block.

Reason 2:

- ✓ If a group of statements is repeatedly required, then it is highly recommended create a function, instead of writing these statements in every time separately.

- ✓ So, it's good to define these statements in a separate block.
- ✓ This block of statements is called as function.
- ✓ Let us understand more by doing practically.

What is a Function?

- ✓ A function contains group of statements which performs the task.

Advantages

- ✓ Maintaining the code is an easy way.
- ✓ Code reusability.

User defined functions

- ✓ Based on requirement, a programmer can create a function.
- ✓ If a programmer created a function, then those functions are called as user defined functions.
- ✓ Now let's understand about user defined functions by doing practically.

Function related terminology

- def keyword
- name of the function
- parenthesis ()
- parameters (if required)
- = (equal symbol)
  - Programmer can create a function with or without = equal symbol.
  - If = equal symbol exists, then that function having return value.
  - If = equal symbol not exists, then that function cannot have return value.
- Function body
- return keyword (optional)

Make a note

✓ After defined a function we need to call the function

Main parts in Function

- ✓ A function can contain mainly two parts,
  - Defining or creating a function
  - Invoking or Calling a function

Defining or creating a function

- ✓ By using def keyword we can create a function.
- ✓ After def keyword we should write name of the function.
- ✓ After function name, we should write parenthesis ()
- ✓ Function body.
  - To perform an operation.
- ✓ Before closing the function, function may contain return type.

Syntax

```
def functionName()
{
        // function body
}
```

Program Name: Define a function
Demo1.scala

```
object Demo1
{
        def main(args: Array[String])
        {
                println("Welcome to main")
        }

        def one()
        {
                println("This is function")
        }
}
```

Compile: scalac Demo1.scala
Run: scalac Demo1.scala

Output

```
Welcome to main
```

Make a note

- ✓ In above program we created a function name as one function.
- ✓ When we execute above program, then that function is not executed because we didn't call the function.
- ✓ So, we need to call that function explicitly to execute function body

2. Calling a function

- ✓ After defining a function, we need to call the function.
- ✓ While calling the function, function name should be match otherwise we will get error.

Program Name: Define a function
Demo2.scala

```
object Demo2
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                one()
        }

        def one()
        {
                println("This is function")
        }
}
```

Compile: scalac Demo2.scala
Run: scalac Demo2.scala

Output

```
Welcome to main
This is function
```

Syntax surprise - 1

- ✓ If a function is not having parameters, then we can ignore parenthesis while calling that function.

Program Name: Define a function
Demo3.scala

```
object Demo3
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                one
        }

        def one()
        {
                println("This is function")
        }
}
```

Compile: scalac Demo3.scala
Run: scalac Demo3.scala

Output

```
Welcome to main
This is function
```

Functions are two types

- ✓ Based on parameters functions are divided into two types,
  - Function without parameters
  - Function with parameters

Function without parameters

- ✓ Functions which have no parameters then that functions are called as a function without parameters.

Program Name: Define a function
Demo4.scala

```
object Demo4
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                empInfo()
        }

        def empInfo()
        {
                println("Welcome to employee information")
        }
}
```

Compile: scalac Demo4.scala
Run: scalac Demo4.scala

Output

```
Welcome to main
Welcome to employee information
```

2. Function with parameters

- ✓ Based on requirement a function can contain parameters.
- ✓ If a function contains parameters, then that functions are called as parameterized functions.
- ✓ These parameters are required to process the function operations.
- ✓ When we pass parameters to function then,
  - Function can capture the parameter's values
  - Perform the operations
  - We will get the result.

Make a note

- ✓ If a function contains parameters, then while calling that function, we need to pass the corresponding values, otherwise we will get error.

Syntax

```
def functionName(paremeter1: Type, parameter2: Type, ….)
{
        // function body
}
```

Program Name: Function which having a parameters
Demo5.scala

```
object Demo5
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                empInfo(23)
        }

        def empInfo(age: Int)
        {
                println("Emp age is: "+age)
        }
}
```

Compile: scalac Demo5.scala
Run: scalac Demo5.scala

Output

```
Welcome to main
Emp age is: 23
```

Make a note:

- ✓ If a function contains parameters, then while calling that function, we need to pass the corresponding values to that function, otherwise we will get error.
- ✓ Below program empInfo having one parameter which is String type, so while calling we should pass String value otherwise we will get error as type mismatch

Program Name: Function which having a parameters
Demo6.scala

```
object Demo6
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                empInfo(23)
        }

        def empInfo(name: String)
        {
                println("Emp name is: "+name)
        }
}
```

Compile: scalac Demo6.scala
Run: scalac Demo6.scala

Output

```
error : type mismatch;
found : Int(11)
required : String
one(11)
```

Program Name: Function which having a parameters
Demo7.scala

```
object Demo7
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                empInfo(23)
        }

        def empInfo(age: Int, name: String)
        {
                println("Emp age is: "+age)
                println("Emp name is: "+name)
        }
}
```

Compile: scalac Demo7.scala
Run: scalac Demo7.scala

Output

```
Welcome to main
Emp age is: 23
Emp name is: Nireekshan
```

A Function can call other function

- ✓ Based on requirement a function can call another function

Program Name: Based on requirement a function can call another function
Demo8.scala

```
object Demo8
{
	def main(args: Array[String])
	{
		println("Welcome to main")
		firstFunction()
	}

	def firstFunction()
	{
		println("This is first function")
		secondFunction()
	}

	def secondFunction()
	{
		println("This is second function")
	}

}
```

Compile: scalac Demo8.scala
Run: scalac Demo8.scala

Output

```
Welcome to main
This is first function
This is second function
```

return keyword in scala

- ✓ return is a keyword in scala programming language.
- ✓ This return keyword we can apply only on functions and methods concept.
- ✓ Based on return statement we can divide functions are two types.
  - Function without return statement
  - Function with return statement.

1. Function without return statement

- ✓ If a function cannot contain return statement, then that function is called as a function without return statement.

Program Name: A function without return statement
Demo9.scala

```
object Demo9
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                balance()
        }

        def balance()
        {
                println("My balance is: ")
        }
}
```

Compile: scalac Demo9.scala
Run: scala Demo9

Output

```
Welcome to main
My balance is:
```

2. Function with return statement

- ✓ Based on requirement a function can contain return statement.
- ✓ The purpose of writing return statement with function is,
  - a function with return statement can return the result.
- ✓ Let's understand by doing practically.
  - Syntactically we can write return statement to function, while creating function with return then we need to use
    - **:** symbol,
    - Type of the value
    - **=** equals symbol

Syntax

```
def functionName(): Type =
{
        // function body

        return value
}
```

If function having return statement then,

- ✓ That function can,
  - Take input,
  - Process it,
  - returns output.

Program Name: A function with return statement
Demo10.scala

```
object Demo10
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                balance()
        }

        def balance(): Int=
        {
                println("My balance is: ")
                return 100
        }
}
```

Compile: scalac Demo10.scala
Run: scala Demo10

Output

```
Welcome to main
My balance is:
```

Important point on return statement

- ✓ If a function contains return statement then while calling that function, that function calling we need to assign to a variable.

Program Name: A function with return statement
Demo11.scala

```
object Demo11
{
        def main(args: Array[String])
        {
                println("Welcome to main")

                var b=balance()

                print(b)
        }

        def balance(): Int=
        {
                println("My balance is: ")
                return 100
        }
}
```

Compile: scalac Demo11.scala
Run: scala Demo11

Output

```
Welcome to main
My balance is:
100
```

Program Name: A function with return statement
Demo12.scala

```
object Demo12
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                println(balance())
        }

        def balance(): Int=
        {
                println("My balance is: ")
                return 100
        }
}
```

Compile	scalac Demo12.scala
Run	scala Demo12
Output	

```
Welcome to main
My balance is:
100
```

Why we need to assign function calling to a variable?

- ✓ So, this assigned variable will be holding the result of function returned value.
- ✓ This variable we can use further in coding.

Program Name: A function with return statement
Demo13.scala

```
object Demo13
{
        def main(args: Array[String])
        {
                println("Welcome to main")

                var b=balance()

                if(b>=0)
                {
                        println(b)
                }

                else
                {
                        println("balance is negative please deposit")
                }
        }

        def balance(): Int=
        {
                println("My balance is: ")
                return 100
        }
}
```

Compile: scalac Demo13.scala
Run: scala Demo13

Output

```
Balance is negative please deposit
```

Program Name: A function with return statement
Demo14.scala

```
object Demo14
{
	def main(args: Array[String])
	{
		println("Welcome to main")

		var b=balance()

		if(b>=0)
		{
			println(b)
		}

		else
		{
			println("Balance is negative please deposit")
		}
	}

	def balance(): Int=
	{
		println("My balance is: ")
		return -123
	}
}
```

Compile: scalac Demo14.scala
Run: scala Demo14

Output

Balance is negative please deposit

return vs Unit type

- ✓ If any function is not return any value, then by default that function returns Unit type.
- ✓ We can also say as, a function which is not having return statement still that function is returning Unit type value.

Program Name: function which having Unit return type
Demo15.scala

```
object Demo15
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                var b=balance()
                print(b)
        }

        def balance()
        {
                println("My balance is: ")
        }
}
```

Compile: scalac Demo15.scala
Run: scala Demo15

Output

```
Welcome to main
My balance is:
()
```

Unit type

- ✓ If any function is not return any value, then by default that function returns Unit type value.
- ✓ So, in this scenario we can assign function return type value as a Unit type, anyway writing Unit type after function name is an optional.

Program Name: function which having Unit return type
Demo16.scala

```
object Demo16
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                var b=balance()
                println(b)

        }

        def balance(): Unit =
        {
                println("My balance is: ")
        }
}
```

Compile: scalac Demo16.scala
Run: scala Demo16

Output

```
Welcome to main
My balance is:
()
```

Syntax surprise - 2

- ✓ Syntactically writing return keyword is an optional in scala programming language.

Program Name: function which having return type
Demo17.scala

```
object Demo17
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                var b=balance()
                println(b)

        }

        def balance(): Int=
        {
                println("My balance is: ")
                return 100
        }
}
```

Compile: scalac Demo17.scala
Run: scala Demo17

Output

```
Welcome to main
My balance is:
100
```

Program Name: function which having return type but return keyword is optional
Demo18.scala

```
object Demo18
{
	def main(args: Array[String])
	{
		println("Welcome to main")
		var b=balance()
		println(b)
	}


	def balance(): Int=
	{
		println("My balance is: ")
		100
	}
}
```

Compile: scalac Demo18.scala
Run: scala Demo18

Output

```
Welcome to main
My balance is:
100
```

Make a note

- ✓ So, we can directly write a value in end of the function without return statement.

Important point about return statement

- ✓ Make sure, function should return corresponding value means,
  - If a function returns type is Int then it should return integer value otherwise we will get error.
  - If a function return type is String, then it should return String value otherwise we will get error

Program Name: A function which is returning String value.
Demo19.scala

```
object Demo19
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                var b=balance()
                println(b)
        }

        def balance(): String=
        {
                println("My balance is: ")
                return "Hello"
        }
}
```

Compile: scalac Demo19.scala
Run: scala Demo19

Output

```
My balance is:
```

Program Name: Error: function return type is Int but returning String value
Demo20.scala

```
object Demo20
{
	def main(args: Array[String])
	{
		println("Welcome to main")
		var b=balance()
		println(b)
	}

	def balance(): Int=
	{
		println("My balance is: ")
		return "Hello"
	}
}
```

Compile: scalac Demo20.scala
Run: scala Demo20

Output

My balance is:

Program Name: Error: function return type is Int but returning String value
Demo21.scala

```
object Demo21
{
	def main(args: Array[String])
	{
		println("Welcome to main")
		var b=balance()
		println(b)
	}


	def balance(): String=
	{
		println("My balance is: ")
		return 100
	}
}
```

Compile: scalac Demo21.scala
Run: scala Demo21

Output

My balance is:

Function Parameters with default Values

A parameterized function can contain default values to the parameters.

- ✓ During function calling if we not passing any values for parameterized function then these values will be assigned.
- ✓ It uses default values of parameters.

Program Name: A parameterized function can contain default values to the parameters.
Demo22.scala

```
object Demo22
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                firstFunction()
        }


        def firstFunction(a: Int=0, b: Int=0)
        {
                println("a value is:"+a)
                println("b value is:"+b)
        }
}
```

Compile: scalac Demo22.scala
Run: scala Demo22

Output

```
My balance is:
```

Make a note:

- ✓ If a function contains default values, during function calling still if we provide values then new values will be replaced with default values.

Program Name: A parameterized function can contain default values to the parameters.
Demo23.scala

```
object Demo23
{
	def main(args: Array[String])
	{
		println("Welcome to main")
		firstFunction(11)
	}

	def firstFunction(a: Int=0, b: Int=0)
	{
		println("a value is:"+a)
		println("b value is:"+b)
	}
}
```

Compile: scalac Demo23.scala
Run: scala Demo23

Output:

```
My balance is:
```

Program Name: A parameterized function can contain default values to the parameters.
Demo24.scala

```
object Demo24
{
	def main(args: Array[String])
	{
		println("Welcome to main")
		firstFunction(11, 22)
	}

	def firstFunction(a: Int=0, b: Int=0)
	{
		println("a value is:"+a)
		println("b value is:"+b)
	}
}
```

Compile: scalac Demo24.scala
Run: scala Demo24

Output:

```
My balance is:
```

Scala Function Named Parameter

- ✓ In parameterized function, during function calling, we can specify the names of parameters also.
- ✓ This concept is called as function named parameters

Program Name: A parameterized function can contain default values to the parameters.
Demo25.scala

```
object Demo25
{
	def main(args: Array[String])
	{
		println("Welcome to main")
		firstFunction(a=11, b=22)
	}

	def firstFunction(a: Int, b: Int)
	{
		println("a value is:"+a)
		println("b value is:"+b)
	}
}
```

Compile: scalac Demo25.scala
Run: scala Demo25

Output

My balance is:

- ✓ In this case, we can pass named parameters in any order and can also pass values only.

Program Name: A parameterized function can contain default values to the parameters.
Demo26.scala

```
object Demo26
{
	def main(args: Array[String])
	{
		println("Welcome to main")
		firstFunction(b=22, a=11)
	}

	def firstFunction(a: Int, b: Int)
	{
		println("a value is:"+a)
		println("b value is:"+b)
	}
}
```

Compile: scalac Demo26.scala
Run: scala Demo26

Output

My balance is:

Scala functions are first class values

- ✓ Scala is a first-class function language.
- ✓ It means,
  - A function can pass another function as a parameter,
  - A function can return another function,
  - Composing function,
  - Nested functions

Higher Order Functions

- ✓ Functions are first class values in scala; means if we create a function then internally it creates a value.

What is higher order function?

- ✓ In scala, a function can take another function as a parameter.
- ✓ In scala, a function can return another function.

- ✓ If a function takes another function as a parameter and returns another function, then that function is called higher order function.

Usage of higher order functions

- ✓ By using higher order functions, we can create,
  - Function composing.
  - Lambda functions.
  - Anonymous function.

Case 1: Passing a Function as Parameter in a Function

- ✓ As discussed in scala, a function can takes another function as a parameter

Program Name: A parameterized function can contain default values to the parameters.
Demo27.scala

```
object Demo27
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                one(22, two(33))
        }


        def one(a: Int, b: Int)
        {
                println("addition of two values:"+(a+b))
        }

        def two(c: Int): Int=
        {
                return c
        }

}
```

Compile: scalac Demo27.scala
Run: scala Demo27

Output

My balance is:

Case 2: A function can return another function

- ✓ We already know regarding return statement like, a function can return a value.
- ✓ Based on requirement a function can return another function.

Program Name: A parameterized function can contain default values to the parameters.
Demo28.scala

```
object Demo28
{
	def main(args: Array[String])
	{
		println("Welcome to main")
		val x=one()
		print(x)
	}


	def one(): Any =
	{
		return two
	}

	def two()
	{
		println("This is from two")
	}

}
```

Compile: scalac Demo28.scala
Run: scala Demo28

Output

My balance is:

Case 3: Function composing

- ✓ A function composition means a function is mixed with another other functions.

Program Name: A parameterized function can contain default values to the parameters.
Demo29.scala

```
object Demo29
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                val x=one(22, two(33))
                println("Addition value is:"+x)
        }

        def one(a: Int, b: Int): Int=
        {
                return a+b
        }

        def two(c: Int): Int=
        {
                return c
        }

}
```

Compile: scalac Demo29.scala
Run: scala Demo29

Output

My balance is:

Case 4: Scala Anonymous (lambda) Function

- ✓ A function which has no name, that function is called as anonymous function.
- ✓ Anonymous function also called as lambda function.

Purpose of anonymous or lambda functions?

- ✓ Just instant use.

Creating anonymous function

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

Program Name: A parameterized function can contain default values to the parameters.
Demo30.scala

```
object Demo30
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                var result = (a: Int, b: Int) => a+b
                var x=result(11, 22)
                println("Addition of a and b values are: "+x)
        }
}
```

Compile: scalac Demo30.scala
Run: scala Demo30

Output

My balance is:

Anonymous function by using _ (underscore) wild card

- ✓ We can create anonymous function by using underscore symbol or wild card.

Program Name	A parameterized function can contain default values to the parameters. Demo31.scala

```
object Demo31
{
        def main(args: Array[String])
        {
                println("Welcome to main")
                var result = (_: Int) +( _: Int)
                var x=result(11, 22)
                println("Addition of a and b values are: "+x)
        }
}
```

Compile	scalac Demo31.scala
Run	scala Demo31
Output	My balance is:

Scala Function Currying

- ✓ By using currying, we can transform a function

Program Name: A parameterized function can contain default values to the parameters.
Demo32.scala

```
object Demo32
{
        def main(args: Array[String])
        {
                println("Welcome to main")

                def add(x: Int, y: Int) = x+y

                println("Addition of x and y values are: "+add(11, 22))
        }
}
```

Compile: scalac Demo32.scala
Run: scala Demo32

Output:

```
My balance is:
```

Program Name: A parameterized function can contain default values to the parameters.
Demo33.scala

```
object Demo33
{
        def main(args: Array[String])
        {
                println("Welcome to main")

                def add(x: Int)(y: Int) = x+y

                println("Addition of x and y values are: "+add(11)( 22))
        }
}
```

Compile: scalac Demo33.scala
Run: scala Demo33

Output:

```
My balance is:
```

Program Name: A parameterized function can contain default values to the parameters.
Demo34.scala

```
object Demo34
{
        def main(args: Array[String])
        {
                println("Welcome to main")

                def add(x: Int)(y: Int) = x+y

                var p=add(11)_
                var q=p(12)
                print(q)
        }
}
```

Compile: scalac Demo34.scala
Run: scala Demo34

Output

My balance is:

Function with Variable Length Parameters

- ✓ If we define a function with two parameters, then during function calling we need to pass two values.
- ✓ If we define a function with three parameters, then during function calling we need to pass three values.
- ✓ If we defined variable length parameterized function, then during function calling we can pass any number of values.
- ✓ We can create variable length parameterized function by using * star symbol.

Program Name: A parameterized function can contain default values to the parameters.
Demo35.scala

```
object Demo35
{
        def main(args: Array[String])
        {
                boyShopping(999, 777)
        }

        def boysShopping(item1: Int, item2: Int)
        {
                print("Total bill is:"+(item1+item2))
        }
}
```

Compile: scalac Demo35.scala
Run: scala Demo35

Output:

```
My balance is:
```

Program Name: A parameterized function can contain default values to the parameters.
Demo36.scala

```
object Demo36
{
        def main(args: Array[String])
        {
                boyShopping(999, 777, 888)
        }

        def boysShopping(item1: Int, item2: Int)
        {
                print("Total bill is:"+(item1+item2))
        }
}
```

Compile: scalac Demo36.scala
Run: scala Demo36

Output:

```
My balance is:
```

Program Name: A parameterized function can contain default values to the parameters.
Demo37.scala

```
object Demo37
{
	def main(args: Array[String])
	{
		boyShopping(999, 777, 888)
	}

	def girlsShopping(item1: Int, item2: Int)
	{
		var sum = 0

		for(a <- items)
		{
			sum=sum+a
		}
		print("Total bill is:"+sum)
	}
}
```

Compile: scalac Demo37.scala
Run: scala Demo37

Output

My balance is:

Program Name: A parameterized function can contain default values to the parameters.
Demo38.scala

```
object Demo38
{
	def main(args: Array[String])
	{
		boyShopping(999, 777, 888,222, 5555, 9876)
	}

	def girlsShopping(item1: Int, item2: Int)
	{
		var sum = 0

		for(a <- items)
		{
			sum=sum+a
		}
		print("Total bill is:"+sum)
	}
}
```

Compile: scalac Demo38.scala
Run: scala Demo38

Output

My balance is:

Nested Functions

- ✓ If we write a function inside another function, then that function is called as nested function.
- ✓ Nested function also called as inner function.
- ✓ If we create a nested function, then we need to call that function in outside function to execute

Program Name: A parameterized function can contain default values to the parameters.
Demo39.scala

```
object Demo39
{
	def main(args: Array[String])
	{
		Outer()
	}

	def outer()
	{
		println("Outer function")

		def nested()
		{
			println("Nested function")
		}
	}
}
```

Compile: scalac Demo39.scala
Run: scala Demo39

Output

My balance is:

Program Name: A parameterized function can contain default values to the parameters.
Demo40.scala

```
object Demo40
{
	def main(args: Array[String])
	{
		outer()
	}

	def outer()
	{
		println("Outer function")

		def nested()
		{
			println("Nested function")
		}

		nested()
	}
}
```

Compile: scalac Demo40.scala
Run: scala Demo40

Output

My balance is:

16. Tuple

Scala Tuples

- ✓ A tuple is a collection of elements in scala
- ✓ It can store same type of elements
- ✓ It can store different type of elements
- ✓ Insertion order is fixed in tuple.

Program Name: Create a tuple
Demo1.scala

```
object Demo1
{
	def main(args:Array[String])
	{
		var t1 = (1, 22, 4, 5, 89)
		var t2 = ("Nireekshan", "Veeru", "Abhi")
		var t3 = (1, 2.5, "Nireekshan")

		println(t1)
		println(t2)
		println(t3)
	}
}
```

Compile: scalac Demo1.scala
Run: scalac Demo1.scala

Output

```
(1,22,4,5,89)
(Nireekshan,Veeru,Abhi)
(1,2.5,Nireekshan)
```

Accessing values from tuple

Program Name: Define a function
Demo2.scala

```
object Demo2
{
	def main(args:Array[String])
	{
		var t = (99, 22, 4, 5, 89)
		println(t._1)
		println(t._2)
	}
}
```

Compile: scalac Demo2.scala
Run: scalac Demo2.scala

Output

```
99
22
```

Program Name: Define a function
Demo3.scala

```
object Demo3
{
        def main(args:Array[String])
        {
                var t = (99, 22, 4, 5, 89)
                t.productIterator.foreach(println)
        }
}
```

Compile: scalac Demo3.scala
Run: scalac Demo3.scala

Output

```
99
22
4
5
89
```

17. Scala Exception Handling

In scala programming we have two kinds of executions,

- ✓ Normal flow of the execution
- ✓ Abnormal flow of execution

Normal flow of the execution

- ✓ In a program if all statements are executed as per the conditions and successfully got output then that flow is called as normal flow of the execution.
- ✓ Below program executed successfully from starting to ending.

Program Name: Normal flow
Demo1.scala

```
object Demo1
{
        def main(args:Array[String])
        {
                println("one")
                println("two")
                println("three")
                println("four")
                println("five")
        }
}
```

Compile: scalac Demo1.scala
Run: scalac Demo1.scala

Output

```
one
two
three
four
five
```

Abnormal flow of the execution

- ✓ While executing statements in a program, if any error occurred at runtime then immediately program flow will get terminates abnormally
- ✓ This kind termination is called as abnormal flow of the execution.

Program Name: Abnormal flow
Demo2.scala

```
object Demo2
{
        def main(args:Array[String])
        {
                println("one")
                println("two")
                println(10/0)
                println("four")
                println("five")
        }
}
```

Compile: scalac Demo2.scala
Run: scalac Demo2.scala

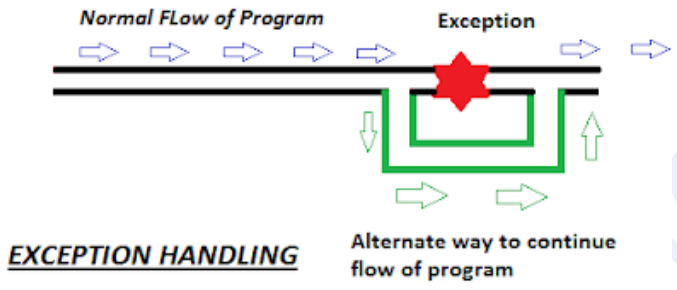
Output

```
one
two
java.lang.ArithmeticException: / by zero
```

- ✓ Above program terminated in middle where run time error got occurred.
- ✓ As discussed, if run time error means it won't execute remaining statements from error onwards.

What we need to do if program terminates abnormally?

- ✓ We need to find an alternative way to finish the work successfully.



What is an Exception?

- ✓ An unwanted, unexpected event which disturbs the normal flow of the program is called exception.
- ✓ When an exception occurred then immediately program will terminate abnormally.
- ✓ We need to handle those exceptions on high priority for normal flow of execution.

Is it really required to handle the exceptions?

- ✓ It is highly recommended to handle exceptions.
- ✓ The main objective of exception handling is for graceful termination of the program(i.e we should not block our resources and we should not miss anything)

What is the meaning of exception handling?

- ✓ Exception handling does not mean repairing exception.
- ✓ We have to define an alternative way to continue rest of the program normally.
- ✓ Defining an alternative way is nothing but exception handling.

Handling exceptions by using try catch

- ✓ We can handle exceptions by using try and catch

try block

- ✓ try is a keyword in python
- ✓ The code which may raise an exception, that code we need to write inside try block.

catch block

- ✓ catch is a keyword in python
- ✓ The corresponding handling code for exception we need write inside catch block.

Make a note

- ✓ try-catch flow:
  - o If any exception raised in try block, then only execution flow goes to catch block for handling code.
  - o If there is no exception, then execution flow won't go to except block

Program Name: try, catch program flow
Demo3.scala

```
object Demo3
{
	def main(args:Array[String])
	{
		println("one")
		println("two")

		try
		{
			println(10/0)
		}

		catch
		{
			case e: ArithmeticException => println("Handling code")
		}

		println("four")
		println("five")
	}
}
```

Compile: scalac Demo3.scala
Run: scalac Demo3.scala

Output

```
one
two
Handling code
four
five
```

try with multiple case blocks inside catch block

- ✓ try with multiple case blocks are allowed, those cases we need to write inside in catch block
- ✓ The way of handling exception is different from exception to exception.
- ✓ So, for every exception type a separate case block we have to write.

Program Name: try, catch program flow
Demo4.scala

```
object Demo4
{
	def main(args:Array[String])
	{
		println("one")
		println("two")

		try
		{
			println(10/0)
		}

		catch
		{
			case a: ArithmeticException => println("Handling code")
			case e: Exception => println("Main exception handling code")

		}

		println("four")
		println("five")
	}
}
```

Compile: scalac Demo4.scala
Run: scalac Demo4.scala

Output

```
one
two
Handling code
four
five
```

Make a note

- ✓ In any project after using all resource its good practice to do clean-up activities.
- ✓ Example: If I've open data base connection after used that connection then I should close that connection.
- ✓ So, a separate place is required to do all clean-up activities.
- ✓ These kinds of activities will be done inside finally block in python

finally block

- ✓ finally, is a keyword in python.
- ✓ We will use finally block to do clean-up activities.

What is the speciality of final block?

- ✓ The speciality of finally block is, it will be **executed always**, irrespective of exception raised or not, exception handled or not

Program Name: try, catch, finally program flow
Demo5.scala

```
object Demo5
{
	def main(args:Array[String])
	{
		println("one")
		println("two")

		try
		{
			println(10/0)
		}

		catch
		{
			case a: ArithmeticException => println("Handling code")
			case e: Exception => println("Main exception handling
code")


		}

		finally
		{
			println("clean up activities like db/file closing")
		}

		println("four")
		println("five")
	}
}
```

Compile: scalac Demo5.scala
Run: scalac Demo5.scala

Output

```
one
two
Handling code
clean up activities like db/file closing
four
five
```

throw keyword or Creating customized exceptions

- ✓ We can create our own exception by using throw keyword

Rules to create customized exception

- ✓ We need to create a class which is super class to Exception class
- ✓ This class should contains String parameterized constructor
- ✓ Based on requirement we need to use throw keyword to create Customized exception object.
- ✓ As per the syntax after throw keyword we need to pass Customized exception class object.

Program Name: Customised exception
Demo6.scala

```
class InvalidAgeException(s:String) extends Exception(s)
{

}

object Demo6
{
        def main(args:Array[String])
        {
                var age = 5

                try
                {
                        if(age<18)
                        {
                                throw new InvalidAgeException("Not eligible")
                        }
                        else
                        {
                                println("You are eligible")
                        }
                }

                catch
                {
                        case e : InvalidAgeException => println("Exception: "+e)

                }
        }
}
```

Compile: scalac Demo6.scala
Run: scalac Demo6.scala

Output

```
Exception: InvalidAgeException: Not eligible
```

Program Name: Define a function
Demo7.scala

```
class InvalidAgeException(s:String) extends Exception(s)
{

}

object Demo7
{
	def main(args:Array[String])
	{
		var age = 50

		try
		{
			if(age<18)
			{
				throw new InvalidAgeException("Not eligible")
			}
			else
			{
				println("You are eligible")
			}
		}

		catch
		{
			case e : InvalidAgeException => println("Exception: "+e)

		}
	}
}
```

Compile: scalac Demo7.scala
Run: scalac Demo7.scala

Output

```
You are eligible
```