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

Feel free to contact if any queries:

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

SCALA INDEX

1. Scala programming introduction

- ✓ What is Scala?
- ✓ Where are all Scala is using?
- ✓ Before Scala, Functional programming was existing but,
- ✓ Before Scala, Object-Oriented Programming also existing but,
- ✓ After Scalia simple definition for FP and OOPs
- ✓ History of Scala?
- ✓ Scala supports
- ✓ Which companies are using scala?
- ✓ Machine language
  - Translator
  - Interpreter
  - Compiler

- ✓ Features of Scala
  - Simple
  - Open source
  - High level language
  - Platform independent
  - Portable
  - Procedure and object oriented
  - Database connectivity

2. Scala keywords and important components in program

- ✓ 99.9999% I'm sure, a scala program contains below things,
- ✓ What is reserved word or keyword?
- ✓ Scala keywords table
- ✓ Keywords count down
- ✓ Scala program structure
- ✓ Access variables, methods and functions

3. Scala coding introduction

- ✓ Ways to write Scala program
- ✓ Scala program execution steps
- ✓ Learn before touch scala program
- ✓ Scala Hello World program
- ✓ Scala Program explanation
- ✓ Scala internal program flow
- ✓ We can also write like below program.

4. Naming conventions in Scala

- ✓ What is identifier?
- ✓ Why should we follow naming conventions?
- ✓ Rules to define identifiers in Scala:
- ✓ Validate the below identifiers
- ✓ Scala program identifiers table
- ✓ Smart suggestions while writing identifiers
- ✓ Comments
- ✓ Types of comments

5. Variables

- ✓ Variable
- ✓ Properties of variable
- ✓ Creating a variables
- ✓ var keyword
  - o What is mutable?
  - o Creating variable by using var
  - o Few points to make a note
  - o When should we go for var variable?
  - o Conclusion for var
  - o Multiple variable initializations
- ✓ val keyword
  - o What is immutable?
  - o Creating variable by using val
  - o Few points to make a note
  - o When should we go for val variable?
  - o Conclusion for val
- ✓ Multiple variable initializations
- ✓ Type inference
- ✓ null value
- ✓ Summary of the story:
- ✓ Invalid cases for variables

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
  - o Byte
  - o Short
  - o Int
  - o 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 char charAt(int index)
  - public String concat(String s)
  - public boolean equals(Object obj)
  - public boolean equalsIgnoreCase(String s)
  - public String substring(int begin)
  - public String substring(int begin , int end)

- public int length()
- public String replace(char old, char new)
- public String toLowerCase()
- public String toUpperCase()
- public String trim()

9. Object Oriented Programming in Scala

- ✓ 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
- ✓ 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?
- ✓ 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?

✓ 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

✓ 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

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.

10. Features of Scala

10.1. Simple

- ✓ Scala syntax is very easy.
- ✓ Developing and understanding Scala is very easy than other.

10.2. Open source

- ✓ We can download freely and customise the code as well

10.3. High level language

- ✓ There are two types of languages,
    - Low level
        - Machine code instructions, difficult to learn programmer.
    - High level
        - English words with syntax, programmer can learn easily.

10.4. Platform independent

- ✓ Scala programs are not dependent on any specific operating systems.
- ✓ We can run on all operating systems happily.

10.5. Portable

- ✓ If a program gives same result on any platform then it is a portable program.
- ✓ Scala used to give same result on any platform.

10.6. Procedure and object oriented

- ✓ Scala supports both procedural and object-oriented features.

10.7. Database connectivity

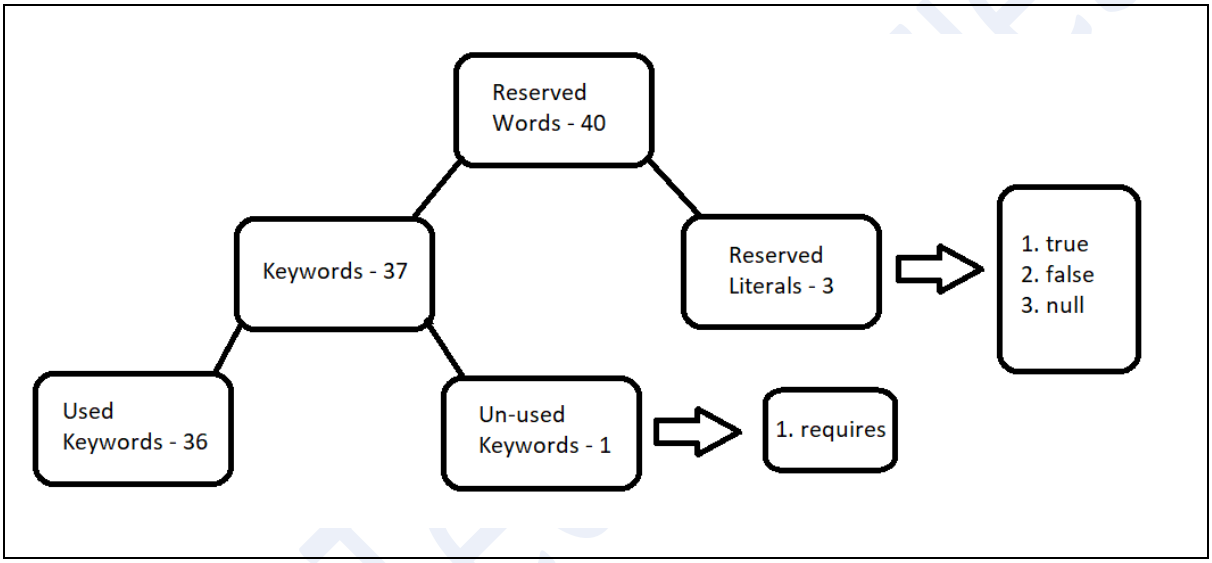
- ✓ Scala provides interfaces to connect with all major databases like, oracle, MySQL, etc...

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

2. Scala program structure

- ✓ Any scala program may contain below things.

1. creating package (by using package keyword)
2. import package(s) (by using import keyword)
3. trait (by using trait keyword)
4. class (by using class keyword)
5. singleton class (by using object keyword)
6. constructor
7. method (by using def keyword)
    1. instance method
    2. singleton method
8. variable (by using val &var keywords)
    1. instance variable
    2. singleton variable
    3. local variable
9. functions (by using def keyword)

3. Access variables, methods and functions

1.	Constructor	:	Automatically executes at the time of object creation
2.	Instance method	:	Need to create object for class to access.
3.	Instance variable	:	Need to create object for class to access.
4.	Singleton method	:	With singleton class name we can access.
5.	Singleton variable	:	With singleton class name we can access.
6.	Local variable	:	We can access directly within the scope.
7.	functions	:	Its, independent we can access directly

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 devolve 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

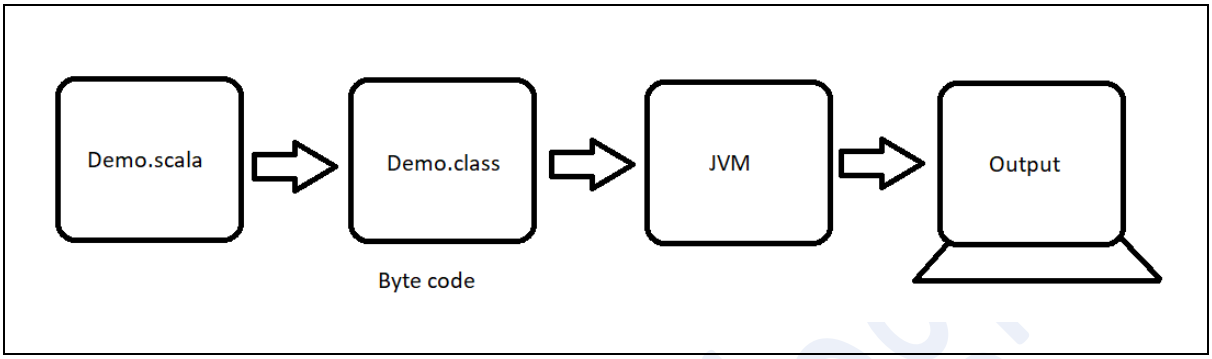
Nireekshan :

- 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
    - label 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

Program Name: checking naming convention
Demo2.scala

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

Compile: scalac Demo2.scala
Run: scala Demo2

Output:

```
Student id is:101
```

Program Name: checking naming convention
Demo3.scala

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

Compile: scalac Demo3.scala
Run: scala Demo3

Output

```
Student id is:101
```

- ✓ If we are using any other symbol like (&,!, -, etc) while defining identifier then we will get compile time error.

Program Name: error: identifier name should not contains illegal symbols
Demo4.scala

```
object Demo4
{
        def main(args: Array[String])
        {
                var student&Id=101
                println("Student id is:"+student&Id)
        }
}
```

Compile: scalac Demo4.scala
Run: scala Demo4

Error

```
Demo4.scala:: error: not found: value &
                        var student&Id=101
                ^
```

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

Program Name: printing variable name which having digits in end of the name
Demo5.scala

```
object Demo5
{
        def main(args: Array[String])
        {
                var studentId123=101
                println("Student id is:"+studentId123)
        }
}
```

Compile: scalac Demo5.scala
Run: scala Demo5

Output

```
Student id is:101
```

Program Name: error: printing variable name which starts with digits, it is invalid
Demo6.scala

```
object Demo6
{
        def main(args: Array[String])
        {
                var 123studentId=101
                println("Student id is:"+123studentId123)
        }
}
```

Compile: scalac Demo6.scala
Run: scala Demo6

Error

```
Demo6.scala:: error: Invalid literal number
                        var 123studentId=101
                            ^
```

3. Identifiers are case sensitive.

Program Name: error: To prove scala is case sensitive
Demo7.scala

```
object Demo7
{
        def main(args: Array[String])
        {
                var a = 101
                println("Value of a is:"+A)
        }
}
```

Compile: scalac Demo7.scala
Run: scala Demo7

Error

```
Demo7.scala:: 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
Demo8.scala

```
object Demo8
{
        def main(args: Array[String])
        {
                var if = 101
                println("Value of a is:"+if)
        }
}
```

Compile: scalac Demo8.scala
Run: scala Demo8

Error

```
Demo8.scala:5: error: illegal start of simple pattern
                var if=101
                    ^
```

5. Spaces are not allowed between identifier.

Program error: space is not allowed while creating an identifier
Name Demo9.scala

```
object Demo9
{
        def main(args: Array[String])
        {
                var student id = 101
                println("Value of a is:"+student id)
        }
}
```

Compile scalac Demo9.scala
Run scala Demo9

Error

```
Demo9.scala:: error: illegal start of simple pattern
                var student id=101
                          ^
```

Summary points

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

✓ If we are using any other symbol like (&, !, -, etc) then we will get syntax error.

2. Identifier allowed digits, but identifier should not start with digit.
3. Identifiers are case sensitive.
4. We cannot use keywords as identifiers.
5. Spaces are not allowed between identifier.

Validate the below identifiers

- ✓ 435student # invalid
- ✓ student564 # valid
- ✓ student565info # valid
- ✓ $tudent # invalid
- ✓ _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

4.5 Smart suggestions while writing identifiers

- ✓ Be descriptive for identifiers,
  - A variable name should describe exactly what it contains.

Program Name: Variable name which is not recommended as per below example
Demo10.scala

```
object Demo10
{
        def main(args: Array[String])
        {
                var a = 101
                println("Student id is:"+a)
        }
}
```

Compile: scalac Demo10.scala
Run: scala Demo10

Output:

```
Student id is: 10
```

Program Name: Variable name which is highly recommended as per below example
Demo11.scala

```
object Demo11
{
        def main(args: Array[String])
        {
                var studentId = 101
                println("Student id is:"+studentId)
        }
}
```

Compile: scalac Demo11.scala
Run: scala Demo11

Output:

```
Student id is: 10
```

- ✓ Don't use abbreviations unnecessarily.
  - Abbreviations may be ambiguous and more difficult to read.

Program Name: Variable name which is highly recommended as per below example
Demo12.scala

```
object Demo12
{
        def main(args: Array[String])
        {
                var fourthBentchMiddleStudentId = 101
                println("Student id is:"+ fourthBentchMiddleStudentId)
        }
}
```

Compile: scalac Demo12.scala
Run: scala Demo12

Output:

```
Student id is: 10
```

Program Name: Variable name which is highly recommended as per below example
Demo13.scala

```
object Demo13
{
        def main(args: Array[String])
        {
                var studentId = 101
                println("Student id is:"+studentId)
        }
}
```

Compile: scalac Demo13.scala
Run: scala Demo13

Output:

```
Student id is: 10
```

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 Demo14.scala

	// This is basic scala program
	object Demo14
	{
	def main(args: Array[String])
	{
	println("Welcome to scala world")
	}
	}

Compile	scalac Demo14.scala
Run	scala Demo14
Output	Welcome to scala world

Program Name	Multi comments in scala program Demo15.scala

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

object Demo15
{
        def main(args: Array[String])
        {
                println("Welcome to scala world")
        }
}
```

Compile	scalac Demo15.scala
Run	scala Demo15
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
```

5.7. Few points to make a note

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

Program Name: Creating variable and reassigning value
Demo4.scala

```
object Demo4
{
	def main(args: Array[String])
	{
		var age=16
		age=18
		println(age)
	}
}
```

Compile: scalac Demo4.scala
Run: scala Demo4

Output

```
18
```

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
Demo5.scala

```
object Demo5
{
        def main(args: Array[String])
        {
                var age=16
                println("My age is sweet: "+age)
        }
}
```

Compile: scalac Demo5.scala
Run: scala Demo5

Output

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

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

6. Multiple variable initializations

- ✓ We can initialize multiple variables together.

Program Name: Creating multiple variables
Demo6.scala

```
object Demo6
{
        def main(args: Array[String])
        {
                var a, b = 10

                println(a)
                print(b)
        }
}
```

Compile: scalac Demo6.scala
Run: scala Demo6

Output

```
10
10
```

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
Demo7.scala

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

Compile: scalac Demo7.scala
Run: scala Demo7

Output

```
101
```

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

```
object Demo8
{
        def main(args: Array[String])
        {
                val empId: Int=101
                println(empId)
        }
}
```

Compile: scalac Demo8.scala
Run: scala Demo8

Output

```
101
```

7.3. Few points to make a note

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

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
                      ^
```

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
Demo10.scala

```
object Demo10
{
        def main(args: Array[String])
        {
                val empId =16
                println("My employee id is:"+ empId)
        }
}
```

Compile: scalac Demo10.scala
Run: scala Demo10

Output

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

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

7.6. Multiple variable initializations

- ✓ We can initialize multiple variables together.

Program Name: Creating multiple variables
Demo11.scala

```
object Demo11
{
        def main(args: Array[String])
        {
                val a, b = 10

                println(a)
                println(b)
        }
}
```

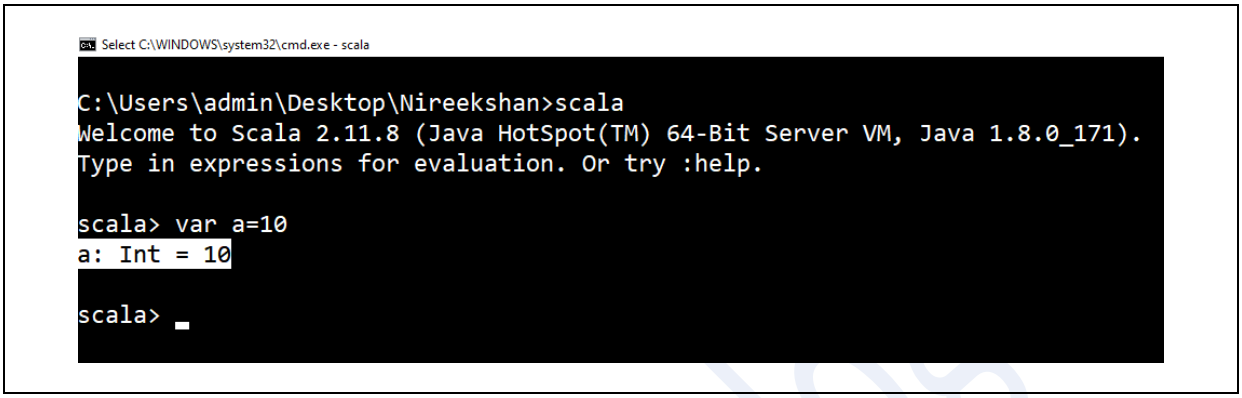
Compile: scalac Demo11.scala
Run: scala Demo11

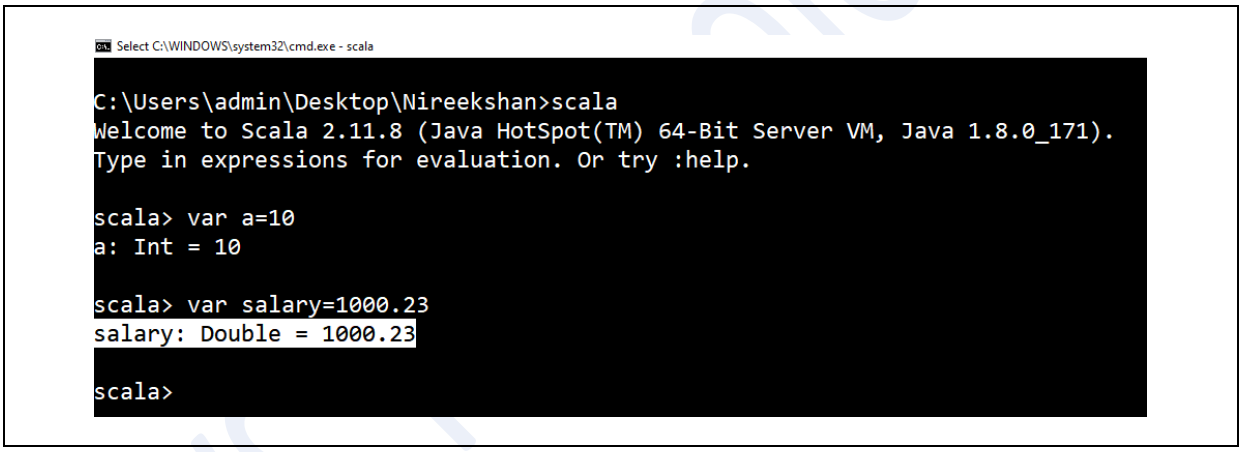
Output

```
10
10
```

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 Demo12.scala

```
object Demo12
{
        def main(args: Array[String])
        {
                val a=null

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

Compile	scalac Demo12.scala
Run	scala Demo12
Output	This variable is holding null value: null

10. Summary of the story:

- ✓ we can create variables by using var and val keywords
- ✓ var variables can be modify
  - var = variable
- ✓ val variables cannot be modify means constants.
  - val = variable + final

11. Invalid cases for variables

1. While defining a variable,
    - Variable name should between in left hand side.
    - Value should be written in right hand side
    - Otherwise we will get compile time error.

Program Name: error: Creating variable in wrong direction
Demo13.scala

```
object Demo13
{
        def main(args: Array[String])
        {
                101 = var a

                println(a)
        }
}
```

Compile: scalac Demo13.scala
Run: scala Demo13

Error

```
Demo3.scala:5: error: ';' expected but '=' found.
                 10 = var a
                    ^
```

2. variables names,
    - should not give as keywords names, otherwise we will get error

Program Name	error: Creating variable in wrong direction Demo14.scala

```
object Demo14
{
        def main(args: Array[String])
        {
                var if = 10

                println(a)
        }
}
```

Compile	scalac Demo14.scala
Run	scala Demo14
Error	

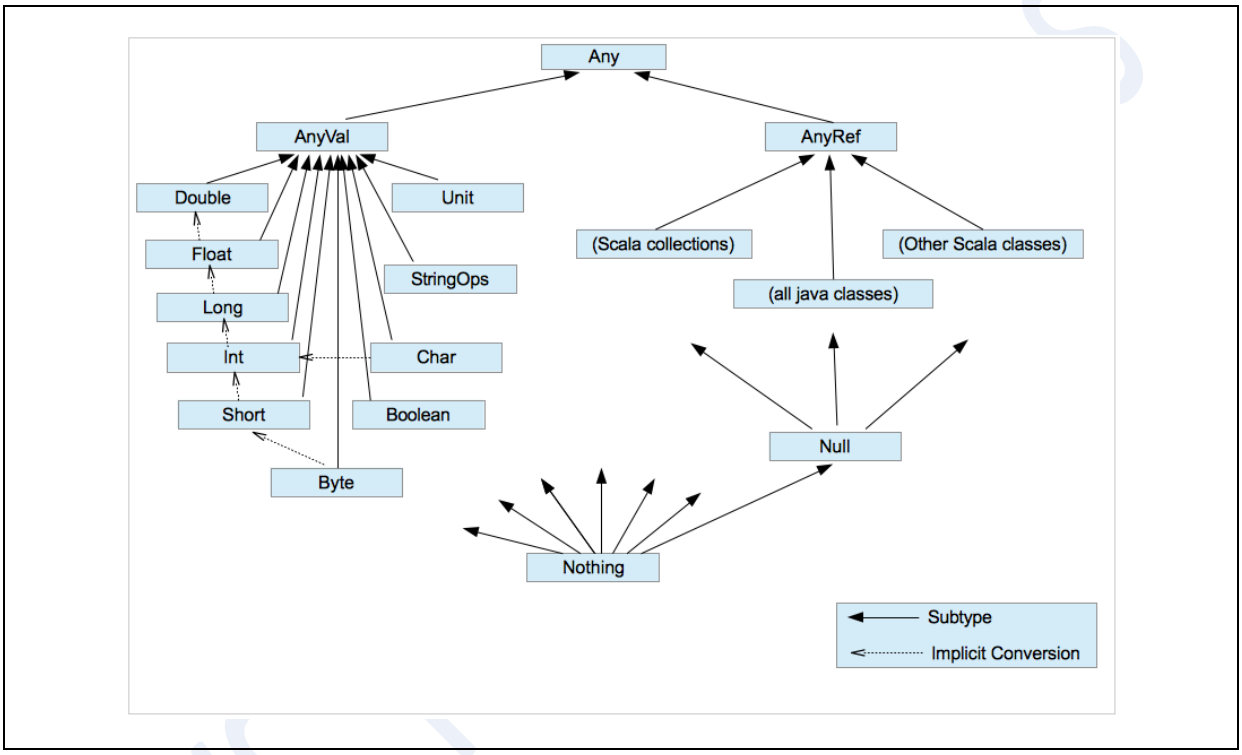
```
Demo14.scala:5: error: illegal start of simple pattern
                var if=10
                    ^
```

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 = 10
                println(x)
        }
}
```

Compile: scalac Demo1.scala
Run: scala Demo1

Output

10

- ✓ Here a is Int represents type of data

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
3. Char data type
    1. Char
4. 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

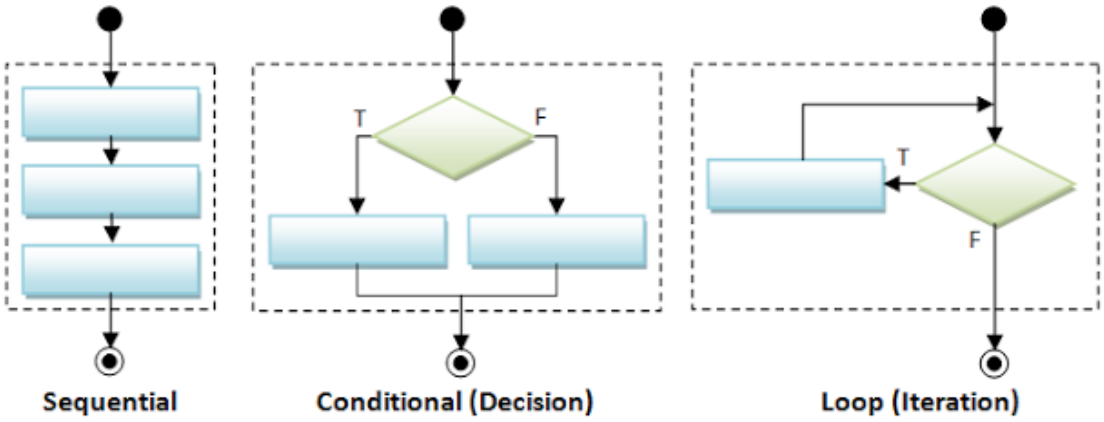
7. Flow control

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

2.1 if
2.2 if else
2.3 if else if
2.4 match

4. Looping

3.1 while
3.2 do while
3.3 for

5. others

4.1 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
```

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

- ✓ **Basically 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:

3. 1. public char charAt(int index)

- ✓ This method returns the character at the specified location index.

Program Name: String charAt example
Demo3.scala

```
object Demo3
{
        def main(args: Array[String])
        {
                var name= new String("Nireekshan")
                println(name)
                println(name.charAt(0))
        }
}
```

Compile: scalac Demo3.scala
Run: scala Demo3

Output

```
Nireekshan
N
```

Program Name: String charAt example
Demo4.scala

```
object Demo4
{
        def main(args: Array[String])
        {
                var name= new String("Nireekshan")
                println(name)
                println(name.charAt(1))

        }
}
```

Compile: scalac Demo4.scala
Run: scala Demo4

Output

```
Nireekshan
i
```

Program Name: error: String charAt example
Demo5.scala

```
object Demo5
{
        def main(args: Array[String])
        {
                var name= new String("Nireekshan")
                println(name)
                println(name.charAt(100))

        }
}
```

Compile: scalac Demo5.scala
Run: scala Demo5

Error:

```
Error: .lang.StringIndexOutOfBoundsException: String index out of range: 100
```

3. 2. 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
```

Program Name: String concat example by using plus symbol
Demo7.scala

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

                println(s1+s2))
        }
}
```

Compile: scalac Demo7.scala
Run: scala Demo7

Output:

```
JavaScala
```

3. 3. 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
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

```
true
false
true
```

Program Name: String equal example
Demo9.scala

```
object Demo9
{
        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 Demo9.scala
Run: scala Demo9

Output

```
false
false
false
```

3.4. public boolean equalsIgnoreCase(String s)

- ✓ This method compares the content of two String objects then it returns the boolean as a result, where the case is not important.

Program Name: String equalsIgnoreCase example
Demo10.scala

```
object Demo10
{
	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 Demo10.scala
Run: scala Demo10

Output

```
true
false
true
```

Program Name: String equalsIgnoreCase example
Demo11.scala

```
object Demo11
{
	def main(args: Array[String])
	{
		var s1= "Java"
		var s2= "java"
		var s3= "Scala"

		println(s1.equalsIgnoreCase(s2))
		println(s1.equalsIgnoreCase(s3))
		println(s1.equalsIgnoreCase(s2))

	}
}
```

Compile: scalac Demo11.scala
Run: scala Demo11

Output

```
true
false
true
```

Make a note:

- ✓ In general to perform validation of user name we have to go for equalsIgnoreCase() method where the case is not importance.
- ✓ Where to perform validations we have to use equals () where the case is important.

3. 5. public String substring(int begin)

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

Program Name: String substring example
Demo12.scala

```
object Demo12
{
        def main(args: Array[String])
        {
                var s1= "Java"

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

Compile: scalac Demo12.scala
Run: scala Demo12

Output

```
ava
```

Program Name: String substring example
Demo13.scala

```
object Demo13
{
        def main(args: Array[String])
        {
                var s1= "Java"

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

Compile: scalac Demo13.scala
Run: scala Demo13

Output

```
va
```

3. 6. public String substring(int begin , int end)

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

Program Name: String substring example
Demo14.scala

```
object Demo14
{
        def main(args: Array[String])
        {
                var s1= "abcdefg"

                println(s1.substring(3))
                println(s1.substring(2,6))

        }
}
```

Compile: scalac Demo14.scala
Run: scala Demo14

Output

```
defg
cdef
```

7. public int length()

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

Program Name: String length example
Demo15.scala

```
object Demo15
{
        def main(args: Array[String])
        {
                var s1= "abcdefg"

                println(s1.length())
                println(s1.substring(2,6))

        }
}
```

Compile: scalac Demo15.scala
Run: scala Demo15

Output

```
7
cdef
```

8. public String replace(char old, char new)

- ✓ This method replaces the all the occurrences of character old by new character

Program Name: String replace example
Demo16.scala

```
object Demo16
{
        def main(args: Array[String])
        {
                var s1= "aaabbb"

                println(s1.replace('a', 'b'))

        }
}
```

Compile: scalac Demo16.scala
Run: scala Demo16

Output

```
bbbbbb
```

9. 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
Demo17.scala

```
object Demo17
{
        def main(args: Array[String])
        {
                var s1= "ABCDEFG"

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

Compile: scalac Demo17.scala
Run: scala Demo17

Output:

```
abcdefg
```

10. 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
Demo18.scala

```
object Demo18
{
        def main(args: Array[String])
        {
                var s1= "abcdefg"

                println(s1.toUpperCase())

        }
}
```

Compile: scalac Demo18.scala
Run: scala Demo18

Output:

```
ABCDEFG
```

11. public String trim()

- ✓ This method removes the blank spaces present at beginning and end of the String, but not blank spaces present at middle of the String

Program Name: String trim example
Demo19.scala

```
object Demo19
{
        def main(args: Array[String])
        {
                var s1 = "      one two three     "
                var s2 = s1.trim()

                println("Before trim method: "+s1)
                println("After trim method: "+s2)

                println("Before trim method string length is: "+s1.length())
                println("After trim method string length is: "+s2.length())
        }
}
```

Compile: scalac Demo19.scala
Run: scala Demo19

Output

```
Before trim method:      one two three
After trim method: one two three
Before trim method string length is: 24
After trim method string length is: 13
```

OOPS

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

Make a note

John :

- ✓ Hey Nireekshan, do I need to follow naming conventions for class while giving name to the class?

Nireekshan :

- ✓ Yes Boss, it's a good practice to follow naming convention while giving names to a class.
- ✓ 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

Nireekshan :

- ✓ 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 😊

Program Name: Creating multiple objects to Student class
Demo4.scala

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

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


object Demo4
{
        def main (args: Array[String])
        {
                val s1 = new Student()
                val s2 = new Student()
                val s3 = new Student()

                s1.display()
                s2.display()
                s3.display()
        }
}
```

Compile
scalac Demo4.scala

Run
scala Demo4

Output

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

Student id is: 101
Student name is: Nireekshan

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

Program Name: Before creating an object class should exists otherwise, we will get error
Demo5.scala

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

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


object Demo5
{
        def main (args: Array[String])
        {
                val e = new Employee()
                e.display()
        }
}
```

Compile: scalac Demo5.scala
Run: scala Demo5

Output

```
error: not found: type Employee
```

Make a note

- ✓ An object exists physically in this world, but class does not exist.
- ✓ An object does not exist without class.
- ✓ A class can exist without an object.

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
Demo6.scala

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

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


object Demo6
{
        def main(args: Array[String])
        {
                val h = new HdfcBank()
                h.bankBalance ()
        }
}
```

Compile: scalac Demo6.scala
Run: scala Demo6

Output

```
500.0
```

Program Name: Data hiding by using private keyword
Demo7.scala

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

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

object Demo7
{
        def main(args: Array[String])
        {
                val h = new HdfcBank()
                h.bankBalance ()
        }
}
```

Compile: scalac Demo7.scala
Run: scala Demo7

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 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
```

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

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

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

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

object Demo9
{
        def main(args: Arrays[String])
        {
                val t = new Test()
                t.m()
        }
}
```

Compile: scalac Demo9.scala

Run: scala Demo9

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
Demo10.scala

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

object Demo10
{
        def main(args: Array[String])
        {
                val t = new Test()
                t.m
        }
}
```

Compile: scalac Demo10.scala
Run: scala Demo10

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
Demo11.scala

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

object Demo11
{
        def main(args: Array[String])
        {
                val t = new Test()
                t.display(10, 20)
        }
}
```

Compile: scalac Demo11.scala
Run: scala Demo11

Output

```
10
20
```

Program Name: Creating parameterised method and accessing by using object
Demo12.scala

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

object Demo12
{
        def main(args: Array[String])
        {
                val t = new Test()
                t.display(10, 20)
        }
}
```

Compile: scalac Demo12.scala
Run: scala Demo12

Output

```
error: type mismatch
found: Int
required: String
```

Program Name: Creating parameterised method and accessing by using object
Demo13.scala

```
class Student
{
        def display(fname: String, lname: String)
        {
                println("First name is: "+fname)
                println("Last name is: "+lname)
        }
}

object Demo13
{
        def main(args: Array[String])
        {
                val s = new Student()
                s.display("Nireekshan ", "Kasagani ")
        }
}
```

Compile: scalac Demo13.scala
Run: scala Demo13

Output

```
First name is: Nireekshan
Last name is: Kasagani
```

Program Name: Creating parameterised method and accessing by using object
Demo14.scala

```
class Student
{
        def display(name: String, age: Int)
        {
                println("Name is: "+name)
                println " `age is: : "+age)
        }
}

object Demo14
{
        def main(args: Array[String])
        {
                val s  =new Student()
                s.display("Nireekshan ", 16)
        }
}
```

Compile: scalac Demo14.scala
Run: scala Demo14

Output

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

Program Name: Creating parameterised method and accessing by using object
Demo15.scala

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

		else
		{
			println(y)
		}
	}
}


object Demo15
{
	def main(args: Array[String])
	{
		val t = new Test()
		t.display(10, 20)
	}
}
```

Compile: scalac Demo15.scala
Run: scala Demo15

Output

```
20
```

Sometimes Method may not 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
Demo16.scala

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

object Demo16
{
        def main(args: Array[String])
        {
                val d = new Demo1()
                println(d.max(10, 20))
        }
}
```

Compile: scalac Demo16.scala
Run: scala Demo16

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
Demo17.scala

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

object Demo17
{
        def main(args: Array[String])
        {
                val b = new Bank()
                b.balance()
        }
}
```

Compile: scalac Demo17.scala
Run: scala Demo17

Output

My balance is:

Program Name: using return type
Demo18.scala

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

object Demo18
{
        def main(args: Array[String])
        {
                val b = new Bank()
                val bal = b. balance()
                print(bal)
        }
}
```

Compile: scalac Demo18.scala
Run: scala Demo18

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
Demo19.scala

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

object Demo19
{
        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 Demo19.scala
Run: scala Demo19

Output

```
Balance is: 100
```

3. Constructors in scala

3.1 Purpose of constructor

- ✓ To initialize the instance variables. (Demo21.scala)

3.2 When constructor will get execute?

- ✓ We no need to call constructor explicitly.
- ✓ Constructor executes automatically at the time of object creation. (Demo22.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
Demo20.scala

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

object Demo20
{
        def main(args: Array[String])
        {
                println("Welcome to main method ")
        }
}
```

Compile: scalac Demo20.scala
Run: scala Demo20

Output

```
Welcome to main method
```

Program Name: Constructor
Demo21.scala

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


object Demo21
{
        def main(args: Array[String])
        {
                val s=new Student()
        }
}
```

Compile: scalac Demo21.scala
Run: scala Demo22

Output

```
Constructor
```

Program Name: Constructor
Demo22.scala

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

object Demo22
{
        def main(args: Array[String])
        {
                val s1 = new Student()
                val s2 = new Student()
        }
}
```

Compile: scalac Demo22.scala
Run: scala Demo22

Output

```
Constructor
Constructor
```

Make a note

- ✓ Developer no needs to call explicitly.
- ✓ At the time of object creation, it 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
Demo23.scala

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


object Demo23
{
        def main(args: Array[String])
        {
                val s=new Student()
        }
}
```

Compile: scalac Demo23.scala
Run: scala Demo23

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
Demo24.scala

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


object Demo24
{
        def main(args: Array[String])
        {
                var e = new Employee("Nireekshan", 16);
        }
}
```

Compile: scalac Demo24.scala
Run: scala Demo24

Output

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

Program Name: Constructor with parameters
Demo25.scala

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


object Demo25
{
        def main(args: Array[String])
        {
                var e = new Employee("Nireekshan", 16);
                e.showDetails()
        }
}
```

Compile: scalac Demo25.scala
Run: scala Demo25

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
Demo26.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 Demo26
{
        def main(args: Array[String])
        {
                var emp = new Employee(101,"Nireekshan",16);
                emp.showDetails()
        }
}
```

Compile: scalac Demo26.scala
Run: scala Demo26

Output

```
Idis: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.

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
Demo27.scala

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

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


object Demo27
{
	def main(args: Array[String])
	{
		val t = new Two()

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

Compile: scalac Demo27.scala
Run: scala Demo27

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
Demo28.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 Demo28
{
        def main(args: Array[String])
        {
                val c = new Child()

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

Compile: scalac Demo28.scala
Run: scala Demo28

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
Demo29.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 Demo29
{
	def main(args: Array[String])
	{
		val c = new Child()

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

Compile: scalac Demo29.scala
Run: scala Demo29

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
Demo30.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 Demo30
{
        def main(args: Array[String])
        {
                val c = new Child()

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

Compile: scalac Demo30.scala
Run: scala Demo30

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
Demo31.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 Demo31
{
	def main(args: Array[String])
	{
		val c = new Child()

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

Compile: scalac Demo31.scala
Run: scala Demo31

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
Demo32.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 Cextends B
{
	var t: Int = 50
	var u: Int = 60

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

object Demo32
{
	def main(args: Array[String])
	{
		val d = new C()

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

Compile scalac Demo32.scala
Run scala Demo32

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
Demo33.scala

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

class B
{
        var i: Int = 10
}

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

class Demo33
{
        def main(args: Array[String])
        {
                val c = new C()
                print(c.i)
        }
}
```

Compile: scalac Demo33.scala
Run: scala Demo33

Output

```
error: ';' expected but ',' found.

class C extends A, B
                ^
one error found
```

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 exists 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
Demo34.scala

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

}


object Demo34
{
        def main(args: Array[String])
        {
                val s=new Sum()

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

Compile: scalac Demo34.scala
Run: scala Demo34

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
Demo35.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 Demo35
{
        def main(args: Array[String])
        {
                val s=new Sum()

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

Compile: scalac Demo35.scala
Run: scala Demo35

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
Demo36.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 Demo36
{
        def main(args: Array[String])
        {
                val s=new Sum()

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

Compile: scalac Demo36.scala
Run: scala Demo36

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
Demo37.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 Demo37
{
        def main (args: Array[String])
        {
                val a = new Addition()

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

Compile: scalac Demo37.scala
Run: scala Demo37

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
Demo38.scala

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

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

object Demo38
{
	def main(args: Array[String])
	{
		val a = new Addition()

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

Compile: scalac Demo38.scala
Run: scala Demo38

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
Demo39.scala

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

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

object Demo39
{
        def main (args: Array[String])
        {
                val a = new Addition()

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

Compile: scalac Demo39.scala
Run: scala Demo39

Output

```
80.12
40.56
```

Can we overload main () method?

- ✓ Yes, we can overload main method but JVM will always search for signature which having like main(args: Array[String])to start program execution.
- ✓ The other user defined main method we need to call explicitly

Program Name: Overloading main method
Demo40.scala

```
object Demo40
{
        def main(args: Array[Int])
        {
                println("Dupe Hero")
        }

        def main(args: Array[String])
        {
                println("Original Hero")

        }
}
```

Compile: scalac Demo40.scala
Run: scala Demo40

Output:

```
Original Hero
```

Program Name: Overloading main method
Demo41.scala

```
object Demo41
{
	def main(a: Array[Int])
	{
		println("Dupe main method with Array of Int")
	}

	def main(args: Array[String])
	{
		println("Original main method")

		val b = Array(1,2,3)
		main(b)
	}
}
```

Compile: scalac Demo41.scala
Run: scala Demo41

Output

```
Original main method
Dupe main method with Array of Int
```

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
Demo42.scala

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

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

object Demo42
{
	def main(args: Array[String])
	{
		val c = new Child()
		c.m1()
	}
}
```

Compile: scalac Demo42.scala
Run: scala Demo42

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
Demo43.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 Demo43
{
        def main(args: Array[String])
        {
                val c = new Child()

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

Compile: scalac Demo43.scala
Run: scala Demo43

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
Demo44.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 Demo44
{
	def main(args: Array[String])
	{
		val c = new Child()

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

Compile: scalac Demo44.scala
Run: scala Demo44

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
Demo45.scala

```
class Commercial
{
	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 Demo45
{
	def main (args: Array[String])
	{
		val c = new Commercial()
		c.calculateBill(100)

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

Compile: scalac Demo45.scala
Run: scala Demo45

Output

```
Commercial Bill amount: 500.0
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
Demo46.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 Demo46
{
	def main(args: Array[String])
	{
		val c = new Child()

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

Compile scalac Demo46.scala
Run scalac Demo46

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
Demo47.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 Demo47
{
        def main(args: Array[String])
        {
                val c = new Child()

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

Compile: scalac Demo47.scala
Run: scala Demo47

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.

abstract class

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
Demo48.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 Demo48
{
	def main(args: Array[String])
	{
		val s = new Sbi()

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

Compile: scalac Demo48.scala
Run: scala Demo48

Output

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

Program Name: Abstract class and child class giving implementation for abstract methods
Demo49.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 Demo49
{
	def main(args: Array[String])
	{
		val s = new Sbi1()

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

Compile
scalac Demo49.scala

Run
scala Demo49

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
Demo50.scala

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

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

	def interest()
}

object Demo50
{
	def main(args: Array[String])
	{
		val s = new Bank()

	}
}
```

Compile: scalac Demo50.scala
Run: scala Demo50

Output

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

abstract variable

- ✓ Abstract class can contain abstract variables which having no initialization.
- ✓ We need to initialize those variables in sub class of abstract class.

Program Name: Abstract variable
Demo51.scala

```
abstract class Bank
{
        var minBalance: Int
}

class Sbi extends Bank
{
        var minBalance: Int = 500

        def balance()
        {
                println("My balance is rupees: "+minBalance)
        }
}

object Demo51
{
        def main(args: Array[String])
        {
                val s = new Sbi()

                s.balance()
        }
}
```

Compile: scalac Demo51.scala
Run: scala Demo51

Output

```
My balance is rupees: 500
```

If you have time,

- ✓ If you have time, then please prepare these below four cases also about abstract class.

Make a note

- ✓ Syntactically all below programs are valid

Case 1

- ✓ abstract class may not contain anything

Program Name: abstract class may not contain anything
Demo52.scala

```
abstract class A
{
        // No methods, no work, be cool...!!! Dude.
}
```

Compile: scalac Demo52.scala
Run: scala Demo52

Output

Case 2

- ✓ abstract class may contain all abstract methods

Program Name: abstract class may contain all abstract methods
Demo53.scala

```
abstract class A
{
        def m1()
        def m2()
        def m3()
}
```

Compile: scalac Demo53.scala
Run: scala Demo53

Output

Case 3

- ✓ abstract class may contain abstract methods and non-abstract methods

Program Name: abstract class may contain abstract methods and non-abstract methods
Demo54.scala

```
abstract class A
{
        def m1()
        {

        }

        def m2()
        def m3()
}
```

Compile: scalac Demo54.scala
Run: scala Demo54

Output

Case 4.

- ✓ abstract class may contain all implemented methods

Program Name: abstract class may contain all implemented methods
Demo55.scala

```
abstract class A
{
        def m1()
        {

        }

        def m2()
        {

        }

        def m3()
        {

        }
}
```

Compile: scalac Demo55.scala
Run: scala Demo55

Output

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
Demo56.scala

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

	def interest()
}


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


object Demo56
{
	def main (args: Array[String])
	{
		val a = new AndhraBank()

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

Compile
scalac Demo56.scala

Run
scala Demo56

Output

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

Program Name: Creating trait and child classes for trait
Demo57.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 Demo57
{
	def main(args: Array[String])
	{
		val d = new TelanganaBankSub1()

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

Compile: scalac Demo57.scala
Run: scala Demo57

Output

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

Program Name: Object creation is not possible for trait
Demo58.scala

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

object Demo58
{
        def main(args: Array[String])
        {
                val d = new A()
        }
}
```

Compile: scalac Demo58.scala
Run: scala Demo58

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
Demo59.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 Demo59
{
	def main(args: Array[String])
	{
		val c = new Customer()

		c.amazonInfo()
		c.amazonShopping()

		c.flipKartInfo()
		c.flipKartShopping()
```

```
                }
        }
```

Compile scalac Demo59.scala
Run scalac Demo59

Output

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

If you have time,

- ✓ If you have time, then please prepare these below four cases also about trait.

Make a note

- ✓ Syntactically all below programs are valid

Case 1

- ✓ trait may not contain anything
- ✓ trait Serializable, this is called as marker trait

Program Name	trait may not contain anything Demo60.scala

```
trait A
{
        // No methods, no work, be cool...!!! Dude.

}
```

Compile	scalac Demo60.scala
Run	scala Demo60
Output	

Case 2

- ✓ trait may contain all abstract methods

Program Name: trait may contain all abstract methods
Demo61.scala

```
trait A
{
        def m1()
        def m2()
        def m3()
}
```

Compile: scalac Demo61.scala
Run: scala Demo61

Output

Case 3

- ✓ trait may contain abstract methods and default methods

Program Name: trait may contain abstract methods and default methods
Demo62.scala

```
trait A
{
        def m1()
        {

        }

        def m2()
        def m3()
}
```

Compile: scalac Demo62.scala
Run: scala Demo62

Output

Case 4.

- ✓ trait may contain all implemented methods

Program Name: trait may contain all implemented methods
Demo63.scala

```
trait A
{
        def m1()
        {

        }

        def m2()
        {

        }

        def m3()
        {

        }
}
```

Compile: scalac Demo63.scala
Run: scala Demo63

Output

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.

Normal class, Singleton object and Standalone class

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
Demo64.scala

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

object Demo64
{
        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 Demo64.scala
Run: scala Demo64

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
Demo65.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 Demo65
{
	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 Demo65.scala
Run: scala Demo65

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 anormal 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
Demo66.scala

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


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

object Demo66
{
        def main(args: Array[String])
        {
                val d = Animal("Dog ")
                val c = Animal("Cat")

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

Compile: scalac Demo66.scala
Run: scala Demo66

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
Demo67.scala

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

object Demo67
{
        def main(args: Array[String])
        {
                val s1 = new Staff("David", 45)
                val s2 = new Staff("David",45)

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

Compile: scalac Demo67.scala
Run: scala Demo67

Output

```
false
```

Program Name: Creating a case class comparing two objects
Demo68.scala

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

object Demo68
{
        def main(args: Array[String])
        {
                val s1 = Staff("David", 45)
                val s2 = Staff("David",45)

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

Compile: scalac Demo68.scala
Run: scala Demo68

Output

```
true
```

Make a note

- ✓ We can create a parameterised constructor.
- ✓ So, these parameters we can declare as either val or var depends requirement
  - val - Getter methods will create automatically
  - var - Getter and Setter methods will create automatically

1. If constructor parameter declared as a val

- ✓ If parameter declared as a val the scala generates only a getter method.
- ✓ As we know val fields are immutable means we cannot change.

Program Name: If declared constructor parameter as val then Demo69.scala

```
class Name(val name: String)

object Demo69
{
        def main(args: Array[String])
        {
                val n = new Name("Prasad")

                println(n.name)
                n.name = "Nireekshan"
        }
}
```

Compile: scalac Demo69.scala
Run: scala Demo69

Output

```
error: reassignment to val
```

2. If constructor parameter declared as a var

- ✓ If parameter declared as a var the scala generates both setter and getter methods.
- ✓ As we know var fields are mutable means we can change means we can set the value here setter methods work.

Program Name: If declared constructor parameter as var then
Demo70.scala

```
class Name(var name: String)

object Demo70
{
        def main(args: Array[String])
        {
                val n = new Name("Prasad")

                println("Before modifying name is: "+n.name)
                n.name = "Nireekshan"
                println("After modifying name is: "+n.name)
        }
}
```

Compile: scalac Demo70.scala
Run: scala Demo70

Output

```
Before modifying name is: Prasad
Before modifying name is: Nireekshan
```

Just couple of minutes…

- ✓ This is Nireekshan and I tried my BEST to make this document without errors.
- ✓ If you find any errors or suggestions then please drop a mail to: nireekshan@gmail.com
- ✓ Thanks for reading

Thanks again😊