Scala programming language

This doc prepared by Nireekshan under Arjun guidelines, reviewed & corrected by Ramesh sir @DVS

Topics Index

1.Scala introduction

2. Variables

3. Data types

4. Flow control

1.Scala introduction

1. 1 A basic introduction about scala

- ✓ Scala is general purpose and high-level programming language.
    - If any programming is using for development, testing and deployment purposes then that is called as general-purpose programming language.
    - Mainly two types of programming languages,
        - High level
            - Human understandable language
        - Low level
            - Machine understandable language

1.2 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.

1.3 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.

1.4 After Scala programming trend got changed

- ✓ 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. (we will learn more in scala program flow)

Make a note

- ✓ To install Scala software, first we need to install Java software

1.5 Where Scala is using?

- ✓ Desktop applications
- ✓ Web applications
- ✓ Database applications
- ✓ Data processing.
- ✓ Data analysis with Spark.
- ✓ Web applications.
- ✓ Machine learning
- ✓ Data Science and etc

1.6 History of Scala?

- ✓ Scala was created by Martin Odersky.
- ✓ Martin Odersky was,

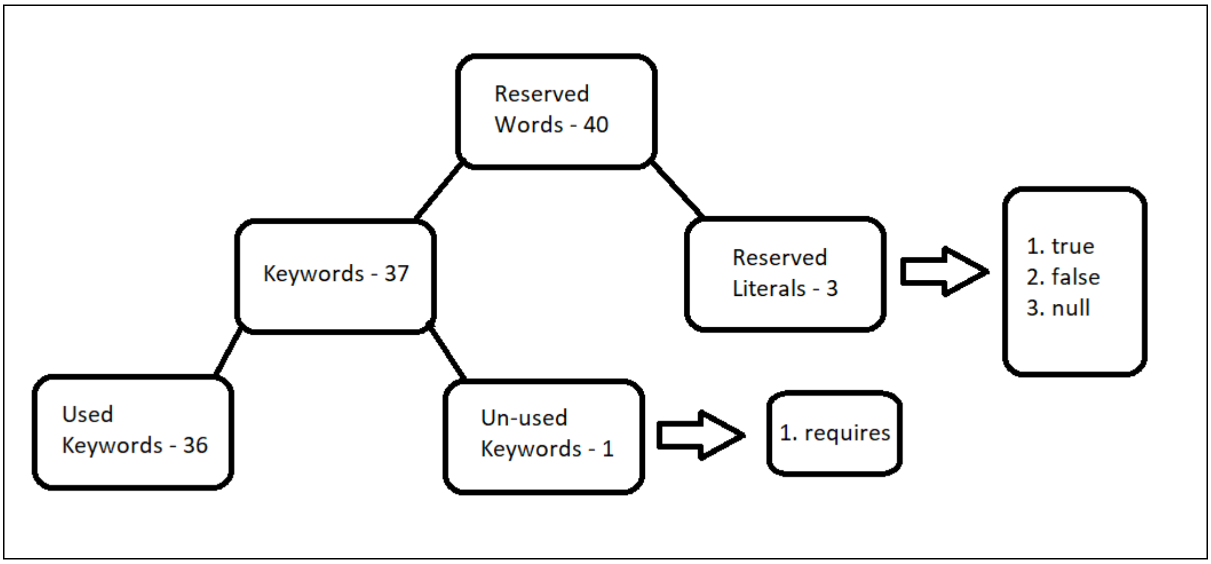
    - Co-designer of Java generics.
    - The original author of the current *javac* reference compiler.

- ✓ Initially first release was in the year of 2004.

1.7 I’m sure 99.9999% a scala program contains below things,

Reserved words or keywords

- ✓ The words which are reserved to do a specific functionality is called as reserved words also called as keywords



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

Scala program structure

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)

How to access variables, methods and functions?

1. Constructor : Automatically executes at the time of object creation

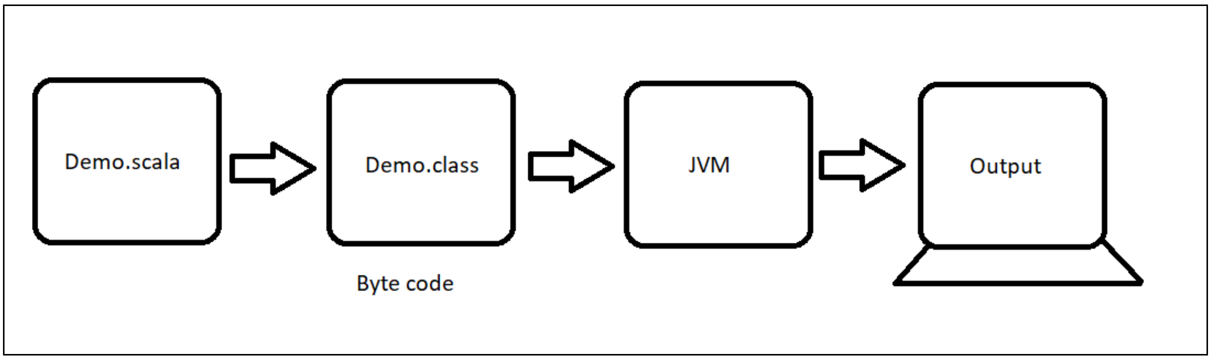
2. instance method : Need to create object for class to access.
3. instance variable : Need to create object for class to access.

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

1.7 Scala internal program flow



- ✓ In very first step we need to write 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.
- ✓ Byte code instructions cannot understandable by the microprocessor to create output.
- ✓ So, the next step is we need to execute this program.
- ✓ To execute this program, we need to use scala command.
  - Example : scala Demo
- ✓ While executing JVM will take responsible to convert byte code instructions into machine understandable format.
- ✓ Then processor will generate output.
  - Welcome to scala world

Compile and run scala program

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

1.8 First scala program

Program Name
Scala hello world program
Demo1.scala

```
object Demo1
{
        def main (args: Array[String])
        {
                println (“Welcome to scala world”)
        }
}
```

Compile
Run

```
scalac Demo1.scala
scala Demo1
```

Output

```
Welcome to scala world
```

Program Name
Scala hello world program
Demo2.scala

```
object Demo2 extends App
{
        println (“Welcome to scala world”)
}
```

Compile
Run

```
scalac Demo2.scala
scala Demo2
```

Output

```
Welcome to scala world
```

Program explanation

- ✓ object

  - It is keyword, by using this keyword we can create singleton class.

- ✓ Program execution starts from 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.

2. Variables in scala

Variable

- ✓ A variable is a,
  - Name
  - refers to a value
  - holds the data
  - name of the memory location.

Purpose of variable

- ✓ To represent values in program

Properties of variable

1. Every variable has a,
   - Name
   - Type
   - Value
   - Scope
   - Location
   - Life time

creating variable

- ✓ Scala provides two keywords to create variables.
  1. var
  2. val

var keyword

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

- ✓ Mutable
  - Once we create a variable by using var then we can re-assign the value to exist variable.

Creating variable by using var

Syntax1

```
var variablename = value
```

Syntax2

```
var variablename: Typeofvariable = value
```

Literal or constant

- ✓ For variables we need to assign value
- ✓ This assigned value also called as literal or constant

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

```
object Demo1
{
        def main (args: Array[String])
        {
                var age=16
                println (age)
        }
}
```

Compile

```
scalac Demo1.scala
```

Run

```
scala Demo1
```

Output

```
16
```

Program   Creating variable by using var keyword
Name   Demo2.scala

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

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

Output

```
16
```

Few points to make a note

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

Program   Creating variable and reassigning value
Name   Demo3.scala

```
object Demo3
{
        def main (args: Array[String])
        {
                var age=16
                age=18
                println (age)
        }
}
```

Compile   `scalac Demo3.scala`
Run   `scala Demo3`

Output

```
18
```

Make a Note

- ✓ We can print meaningful text message along with variable for better understanding
  - ○ Text message we should write within double quotes.
  - ○ Text message and variable name should be separated by plus (+) symbol

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

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

Compile

```
scalac Demo4.scala
```

Run

```
scala Demo4
```

Output

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

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.

Conclusion

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

Multiple variable initialization

- ✓ we can initialize multiple variables together.

Program Name   Creating multiple variables
Demo5.scala

```
object Demo5
{
        def main (args: Array[String])
        {
                var a, b = 10

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

Compile   scalac Demo5.scala

Run   scala Demo5

Output

```
10
10
```

val keyword

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

- ✓ Immutable:
  - Once we initialize a variable by using val then we cannot re-assign the value to that variable.

- ✓ A val is like a final variable in java

Creating variable by using val keyword

Syntax1

```
val variablename = value
```

Syntax2

```
val variablename: Typeofvariable = value
```

Program Name

Creating variable by using val keyword
Demo6.scala

```
object Demo6
{
        def main (args: Array[String])
        {
                val mailid ="nirekshan@gmail.com";
                println(mailid)
        }
}
```

Compile
Run

```
scalac Demo6.scala
scala Demo6
```

Output

```
nirekshan@gmaii.com
```

Program Creating variable by using val keyword
Name Demo7.scala

```
object Demo7
{
        def main (args: Array[String])
        {
                val mailid: String =”nirekshan@gmail.com”;
                println(mailid)
        }
}
```

Compile
```
scalac Demo7.scala
```

Run
```
scala Demo7
```

Output
```
nirekshan@gmaii.com
```

Few points to make a note

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

Program Reassignment is not possible for val variable
Name Demo8.scala

```
object Demo8
{
        def main (args: Array[String])
        {
                val mailid =”nirekshan@gmail.com”;
                mailid=”ramesh@gmail.com”;

                println(mailid)
        }
}
```

Compile scalac Demo8.scala
Run scala Demo8

Error

Make a Note

- ✓ We can print meaningful text message along with variable for better understanding
    - ○ Text message we should write within double quotes.
    - ○ Text message and variable name should be separated by plus (+) symbol

Program Creating variable by using val keyword
Name Demo9.scala

```
object Demo9
{
        def main (args: Array[String])
        {
                val mailid: String =”nirekshan@gmail.com”;
                println(“My mail id is: ”+mailid)
        }
}
```

Compile scalac Demo9.scala
Run scala Demo9

Output

```
My mail id is: nirekshan@gmaii.com
```

When should we go for val variable?

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

Conclusion

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

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

null value

- ✓ While creating a variable we can assign a value as null
- ✓ null value of the variable indicates as that variable or object is empty means nothing

Program Name

Creating variable and assigning with null value
Demo10.scala

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

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

Compile
Run

```
scalac Demo10.scala
scala Demo10
```

Output

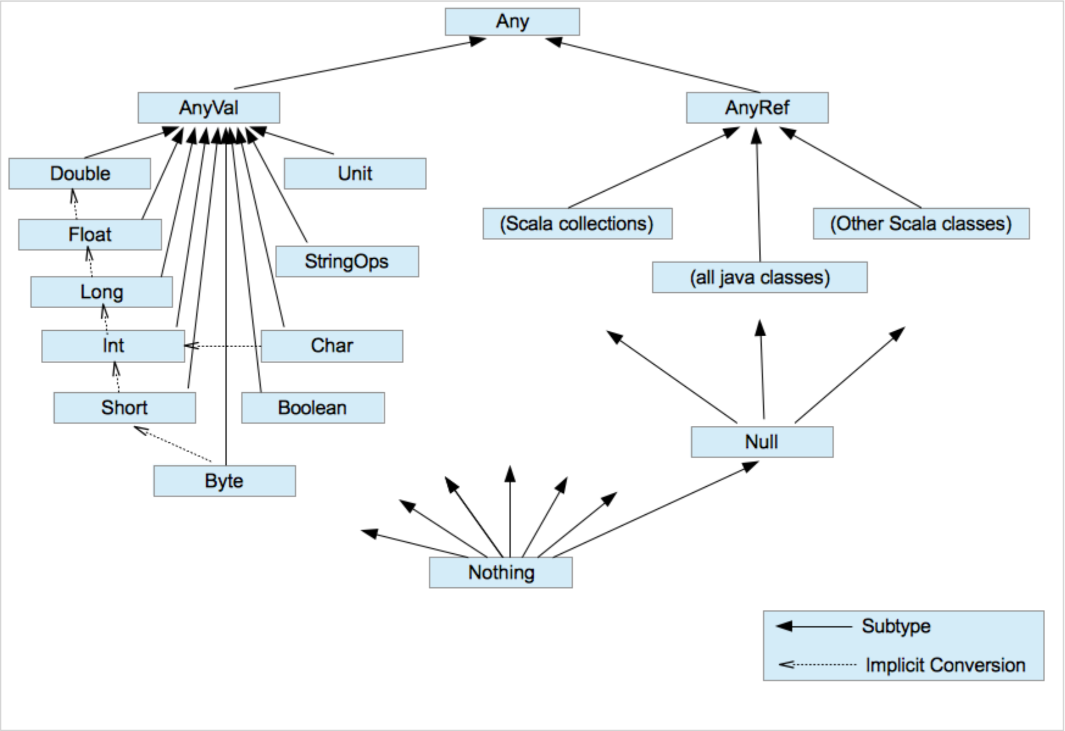
```
Addition of two values is: 30
```

Summary of the story:

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

3 Data types in scala

Scala data types image



Data type

- ✓ A data type represents the type of the data.

Program Name

Creating 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

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

Mostly usage classes from scala package

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

5. scala.Float
6. scala.Double

7. scala.Char

8. scala.Boolean

Types of data types

- ✓ There are three type of data types.

    1. Numeric data types

        1. Integral data types

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

    2. Floating float

        1. Float
        2. Double

    3. char data type

        1. Char

    4. boolean data type

        1. Boolean

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

1. Byte data type

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

Program Name — Creating variable by using val keyword
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;
```

2. Short

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

Program Creating variable by using val keyword

Name Demo3.scala

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

        }
}
```

Compile scalac Demo3.scala

Run scala Demo2

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;
```

3. Int

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

Program Creating variable by using val keyword
Name 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;
```

4. Long

Size : 8 bytes

Program Creating variable by using val keyword
Name 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;
```

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

1. Float

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

  Size : 4 bytes

Program Creating variable by using val keyword
Name Demo6.scala

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

        }
}
```

Compile scalac Demo6.scala
Run scala Demo6

Output

```
10000
```

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;
```

2. Double

Size : 8 bytes

Program Name Creating variable by using val keyword
Demo7.scala

```
object Demo5
{
        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;
```

2.3.3. 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　　Creating variable by using val keyword
Name　　　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;
```

2.3.4. 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 Creating variable by using val keyword
Name Demo9.scala

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

        }
}
```

Compile scalac Demo9.scala
Run scala Demo9

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;
```

Summary

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

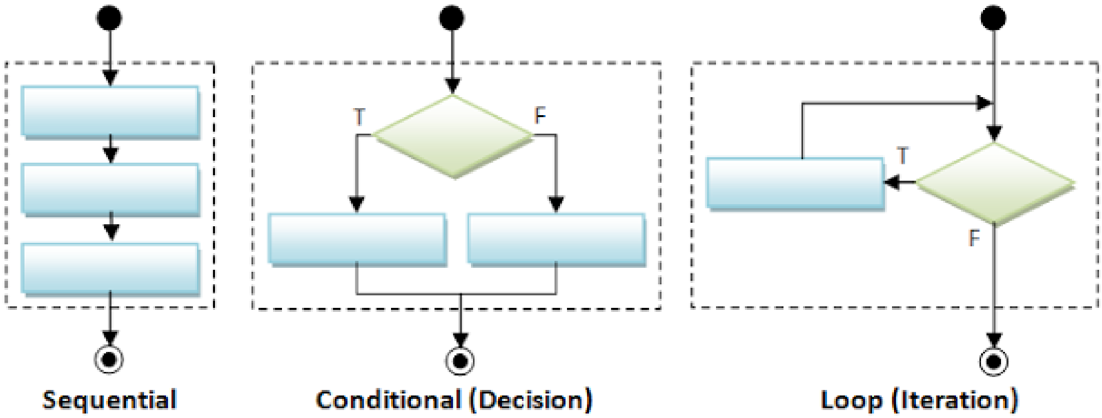
8. Flow control

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.

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.



1. Sequential

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

2. Conditional

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

3. Looping

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

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
Run

```
scalac Demo1.scala
scala Demo1
```

Output

```
One
two
three
four
```

2. Conditional or Decision-making statements

2.1 if
2.2 if else
2.3 if else if
2.4 match

3. Looping

3.1 while
3.2 do while
3.3 for

4. others

4.1 return

2.1 ***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.

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 Basic program on if statement
Name Dem2.scala

```
object Dem2
{
    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
Run

```
scalac Dem2.scala
scala Dem2
```

output

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

Program Basic program on if statement
Name Dem3.scala

```
object Dem3
{
    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
Run

```
scalac Dem3.scala
scala Dem3
```

output

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

2.2 ***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.

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
Run

```
scalac Demo4.scala
scala Demo4
```

output

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

Program Basic program on if else statement
Name 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
Run

```
scalac Demo5.scala
scala Demo5
```

output

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

2.2 ***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.

When should we use if statement?

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

Program Basic program on if else if statement
Name 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
```

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

3. Looping

3.1 do while
3.2 while
3.3 for

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

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

                do
                {
                        println(counter)
                        counter = counter + 1

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

Compile
Run

```
scalac Demo7.scala
scala   Demo7
```

Output

```
1
2
3
4
5
```

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

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

                do
                {
                        println(counter)
                        counter = counter + 1

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

Compile
Run

```
scalac Demo8.scala
scala   Demo8
```

Output

```
1
```

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

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

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

Compile
Run

```
scalac Demo9.scala
scala Demo9
```

output

```
1
2
3
4
5
```

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

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

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

Compile
Run

```
scalac Demo10.scala
scala Demo10
```

output

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 Example using for loop
Name Demo11.scala

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

Compile scalac Demo11.scala
Run scala Demo11

output

```
1
2
3
4
5
```

Syntax

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

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 Example using for loop
Name 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
```

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	Example using for loop
Name	Demo13.scala

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

Compile	scalac Demo13.scala
Run	scala Demo13

output

```
1
3
5
7
9
```

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 `scala Demo14`

output

```
2
4
6
8
10
```

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
```

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
```