

What is Scala:

- 1) Scala is a **general-purpose** programming language that combines concepts of **object-oriented** and **functional** programming languages.
- 2) It supports **object-oriented**, **functional** and **imperative** programming approaches.
- 3) It is a strong **static** type language.
- 4) In Scala everything is an **object** whether it is a function or a number. It does not have concept of **primitive** data types.
- 5) It was designed by **Martin Odersky** and officially released for java platform in early 2004 and for .Net framework in June 2004. Scala dropped .Net support in 2012.
- 6) File extension of scala source file can be either **.scala** or **.sc**

History of Scala:

Scala is a **general-purpose** programming language. It was created and developed by **Martin Odersky**. Martin started working on Scala in **2001** at the **Ecole Polytechnique Federale de Lausanne (EPFL)**. It was officially released on **January 20, 2004**.

Scala was designed to be both **object-oriented** and **functional**. It is a pure object-oriented language in the sense that every **value** is an **object** and functional language in the sense that every **function** is a **value**.

The name of Scala is derived from the word **scalable** which means it can grow with the demand of users.

Scala Characteristics:

- 1) It is an **object-oriented** language that **supports** many **traditional design patterns** being **inherited** from existing programming languages.
- 2) It supports **functional** programming that **enables** it to handle **concurrency** and **distributed** programming.
- 3) Scala is designed to run on **JVM** platform that helps in using **Java libraries** and other **rich feature APIs**.
- 4) Scala is **statically** typed that prevents it from problems of **dynamically** typing.
- 5) Scala is **easy** to implement in **existing** java projects as **Scala libraries** can be used within **Java code**.
- 6) There is **no** need to declare **variables** in Scala as Scala compiler can **infer** most **types** of variables.
- 7) Scala is **not** an extension of Java, but it is completely **inter-operable** with it. While compilation Scala file translates to Java byte code and runs on JVM.

Basic Info:

When we consider a Scala program, it can be defined as a **collection of objects** that communicate via invoking each other's methods. Let us now briefly look into what do **class, object, methods** and **instance variables** mean.

- **Object:** Objects have **states** and **behaviors**. An object is an **instance** of a **class**. **e.g.** A dog has **states**: color, name, breed as well as **behaviors**: wagging, barking, and eating.
- **Class:** **Class** can be defined as a **template/blueprint** that describes the **state/behavior** that are related to the class.
- **Method:** **Method** is basically a **behavior**. A class can contain **many** methods. It is in methods where the **logics** are written, **data is manipulated** and all the **actions** are **executed**.
- **Fields:** Each **object** has its **unique** set of **instance** variables, which are called **fields**. An object's **state** is created by the **values** assigned to these fields.

The following are the **basic coding conventions** in Scala programming.

- **Case Sensitivity:** Scala is **case-sensitive**, which means identifier **Hello** & **hello** would have different meaning in Scala.
- **Class Names:** For all class names, the first letter should be in **Upper Case**. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case. **e.g. class MyFirstScalaClass**
- **Method Names:** All method names should start with a **Lower-Case** letter. If multiple words are used to form the name of the method, then each inner word's first letter should be in Upper Case. **e.g. def myMethodName ()**
- **Program File Name:** Name of the program file should **exactly** match the **object** name. When saving the file, you should save it using the **object** name (**Remember Scala is case-sensitive**) and append **' .scala/.sc'** to the end of the name. If the file name and the object name **do not** match your program will not compile. **e.g.** Assume **'HelloWorld'** is the object name. Then the file should be saved as **'HelloWorld.scala'**.
- **def main (args: Array[String]):** Scala program processing starts from the **main ()** **method** which is a mandatory part of every Scala Program.

First Program in Scala:

We can execute a Scala program in two modes: one is **interactive** mode and another is **script** mode.

1) Interactive Mode:

Open the command prompt and use the following command to open Scala.

```
C:\Users\user>scala_
```

If Scala is installed in your system, the following output will be displayed

```
C:\Users\user>scala
Welcome to Scala version 2.10.6 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_91).
Type in expressions to have them evaluated.
Type :help for more information.
```

Type the following text to the right of the Scala prompt and press the Enter key

```
scala> println("Hello, Scala!")_
```

It will produce the following result

```
Hello, Scala!
```

2) Script Mode:

In the following code, we will create an object **HelloWorld**. It contains a **main** method and display message using **println** method.

a) This file is saved with the name **HelloWorld.scala**

b) Command to compile this code is **scalac HelloWorld.scala**

c) Command to execute the compiled code is **scala HelloWorld**

After executing code, it will yield the output.

Use the following instructions to write a Scala program in script mode. Open notepad and add the following code into it. Save the file as: **HelloWorld.scala**

e.g.

```
object HelloWorld {
def main(args: Array[String]) {
println ("Hello World!")
}
}
```

Open the command prompt window and go to the directory where the program file is saved.




```
C:\Users\user>cd C:\Users\user\Desktop\Spark_Scala_
```

Use the following command to compile and execute your Scala program.

```
C:\Users\user\Desktop\Spark_Scala>scalac HelloWorld.scala_
```

The '**scalac**' command is used to compile the Scala program and it will generate a few class files in the current directory. One of them will be called **HelloWorld.class**. This is a **bytecode** which will run on **Java Virtual Machine (JVM)** using 'scala' command.

C:\Users\user\Desktop\Spark_Scala

| Name | Date modified | Type | Size |
|--|------------------|------------|------|
|  HelloWorld\$.class | 24-12-2018 12:34 | CLASS File | 1 KB |
|  HelloWorld.class | 24-12-2018 12:34 | CLASS File | 1 KB |
|  HelloWorld.scala | 24-12-2018 12:28 | SCALA File | 1 KB |

Then, execute below command

```
C:\Users\user\Desktop\Spark_Scala>scala HelloWorld_
```

Error:

```
C:\Users\user\Desktop\Spark_Scala>scala HelloWorld
No such file or class on classpath: HelloWorld
```

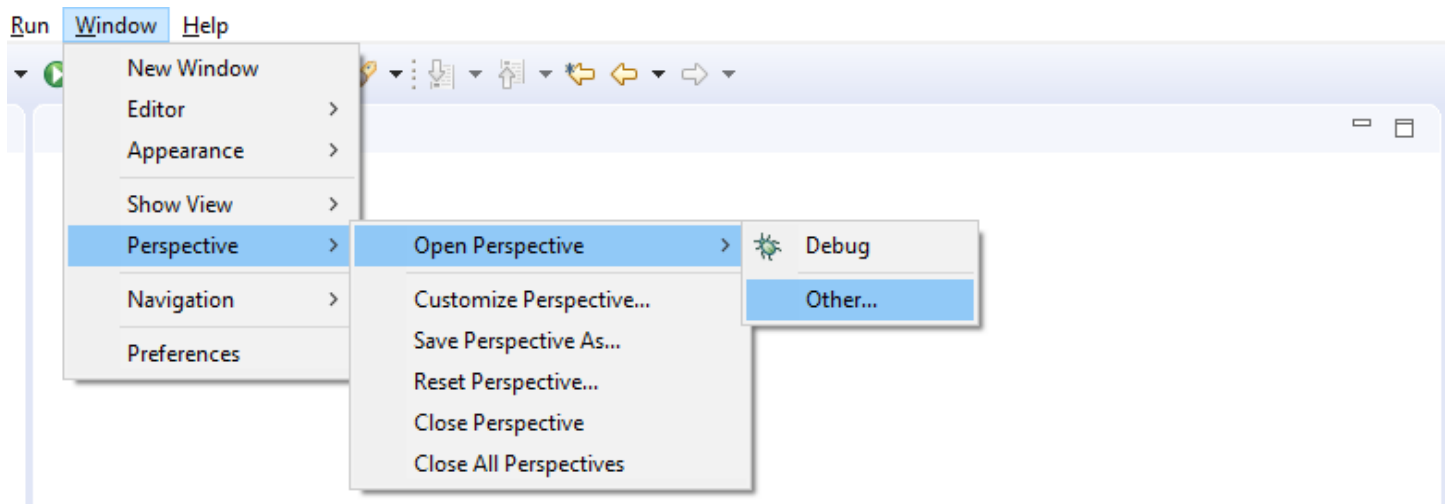
Resolution: The current directory is typically not in the **CLASSPATH** by default. When you have the CLASSPATH variable set, scala will not include the current directory in its search, *you must explicitly add it*. So, you need to give explicitly as below command.

Output:

```
C:\Users\user\Desktop\Spark_Scala>scala -cp . HelloWorld
Hello World!
```

Eclipse:

To create a "**Hello World**" application with **Eclipse**. Assuming we have installed the Scala plugin we can switch to the Scala perspective (*Window* → *Perspective* → *Open Perspective* → *Other* → *Scala* then click "*OK*").

**Step 1:** Create a new Scala project "**Scala_Practice**"

- Go to File → New → Scala Project
- Enter Scala_Practice in the Project name field
- Click the "Finish" button

Step 2: Create a new Scala package in source folder "**src**"

- Right-click on the Scala_Practice project in the "Package Explorer" tab of the projects pane
- Select the New → Package
- In the "New Package" window, enter "scala_handson" in the Name field (**Note:** Package name starts with a lower-case)
- Press the Finish button

Step 3: Create a Scala object HelloWorld with a main method

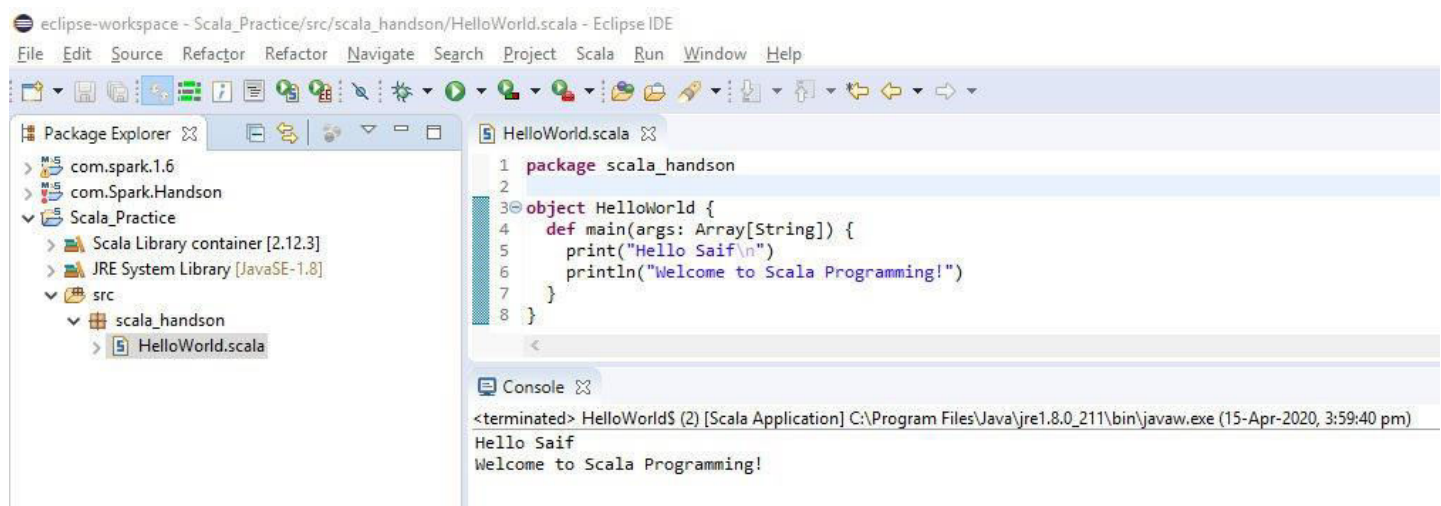
- First expand the "Scala_Practice" project tree and right-click on the "scala_handson" package under "src" folder.
- Select the New → Scala Object and enter "HelloWorld" in the Object name field.
- Press the Finish button

Step 4: Extend the code to print a message. Make the HelloWorld object implement the main method and add a println statement

```
object HelloWorld {  
  def main (args: Array [String]) {  
    //print ("Hello, Saif")           (// as Single Line Comment & \n as New Line)  
    print ("Hello, Saif\n")  
    println ("Welcome to Scala Programming")  
  }  
}
```

Step 5: Create a Run configuration for the Scala project

- Save & Right click on HelloWorld.scala in the Package Explorer
- Select "Run as ..." and select "Scala Application"
- Select the first matching item, the "HelloWorld" class
- Click the "Ok" button



Scala Variables:

Variables are nothing but **reserved memory** locations to **store** values. This means that when you create a variable, you **reserve** some **space** in **memory**.

Based on the **data type** of a variable, the **compiler allocates memory** and decides what can be **stored** in the **reserved memory**. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

Variable Declaration:

Scala has different syntax for declaring variables. They can be defined as **value**, **i.e. constant or a variable**. Here, **myVar** is declared using the keyword **var**. It is a variable that can change value and this is called **mutable** variable. Following is the syntax to define a variable using **var** keyword

Syntax:

var myVar = <Value>

→ **Method 1:** Without assigning data type

```
scala> var a = 10
a: Int = 10

scala> a = 20
a: Int = 20

scala> a = "Saif"
<console>:8: error: type mismatch;
 found   : String("Saif")
 required: Int
    a = "Saif"
      ^
```

var myVar: <Data Type> = <Value>

→ **Method 2:** With assigning data type

```
scala> var myVar:String = "My name is Saif"
myVar: String = My name is Saif

scala> var myVar:String = "I am learning Scala Programming"
myVar: String = I am learning Scala Programming

scala> var myVar:String = 10
<console>:7: error: type mismatch;
 found   : Int(10)
 required: String
    var myVar:String = 10
                        ^
```

Note: You can reassign a new value to **myVar** because it is **mutable**, but you cannot reassign the **variable** to a different **type**.

Eclipse:

Method 1: Declaring Var without Data Type

```

1 package scala_practice
2
3 object Mutable_Var {
4   def main (args: Array [String]) {
5     //Method 1: Declaring Var without Data Type
6     var myVar = "My name is Saif"
7     println (myVar)
8     myVar = "I am learning Scala Programming"
9     println (myVar)
10    myVar = 10
11    println
12  }
13 }

```

type mismatch; found : Int(10) required: String

<terminated> Mutable_Var\$ [Scala Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.exe (24-Dec-2018, 8:44:40 PM)

My name is Saif
I am learning Scala Programming

Method 2: Declaring Var with Data Type

```

1 package scala_practice
2
3 object Mutable_Var {
4   def main (args: Array [String]) {
5     //Method 2: Declaring Var with Data Type
6     var myVar1:String = "My name is Saif"
7     println (myVar1)
8     myVar1 = "I am learning Scala Programming"
9     println (myVar1)
10    myVar1 = 10
11    println
12  }
13 }

```

type mismatch; found : Int(10) required: String

<terminated> Mutable_Var\$ [Scala Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.exe (24-Dec-2018, 8:44:40 PM)

My name is Saif
I am learning Scala Programming

Here, **myVal** is declared using the keyword **val**. This means that it is a variable that cannot be changed and this is called **immutable** variable. Following is the syntax to define a variable using **val** keyword

Syntax:

val myVal = <Value>

→ **Method 1:** Without assigning data type

```
scala> val myVal = "My name is Saif"
myVal: String = My name is Saif

scala> myVal = "I am learning Scala Programming"
<console>:8: error: reassignment to val
      myVal = "I am learning Scala Programming"
          ^
```

val myVal: <Data Type> = <Value>

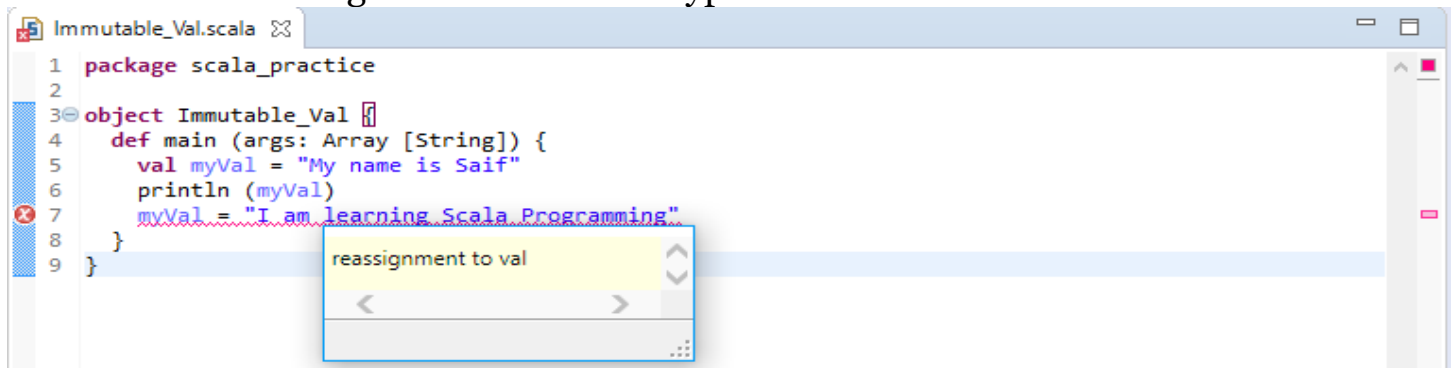
→ **Method 2:** With assigning data type

```
scala> val myVal:String = "My name is Saif"
myVal: String = My name is Saif

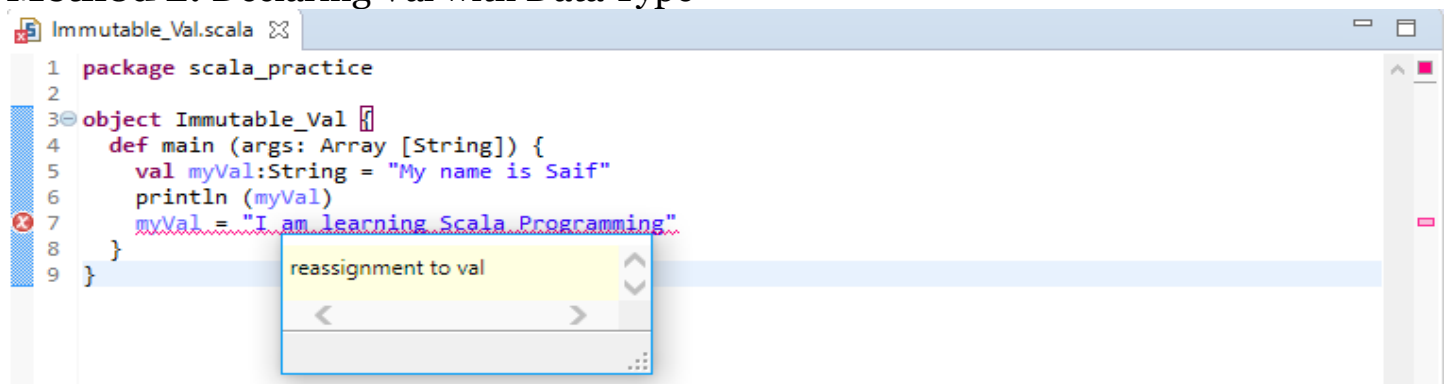
scala> myVal = "I am learning Scala Programming"
<console>:8: error: reassignment to val
      myVal = "I am learning Scala Programming"
          ^
```

Eclipse:

Method 1: Declaring Val without Data Type



Method 2: Declaring Val with Data Type



Variable Type Inference:

When you assign an **initial** value to a **variable**, the Scala compiler can figure out the **type** of the **variable** based on the **value** assigned to it. This is called **variable type inference**. Therefore, you could write these variable declarations like this

Syntax:

```
var myVar = 10
```

```
scala> var myVar = 10
myVar: Int = 10
```

```
val myVal = "This is an e.g. of Variable Type Inference"
```

```
scala> val myVal = "This is an e.g. of Variable Type Inference"
myVal: String = This is an e.g. of Variable Type Inference
```

Note: Here, by default, **myVar** will be **Int type** and **myVal** will become **String type** variable.

Multiple assignments:

Scala supports multiple assignments. If a code block or method returns a Tuple (Tuple – Holds collection of Objects of different types), the Tuple can be assigned to a val variable. [Note: We will study Tuples in subsequent chapters.]

Method 1: Without assigning Data Type

```
scala> val (myVar1, myVar2) = Pair (10, "Hello Saif")
myVar1: Int = 10
myVar2: String = Hello Saif
```

```
scala> val (myVar1, myVar2) = (10, "Hello Saif")
myVar1: Int = 10
myVar2: String = Hello Saif
```

Method 2: With assigning Data Type

```
scala> val (myVar1: Int, myVar2: String) = Pair(40, "Hello Saif")
myVar1: Int = 40
myVar2: String = Hello Saif
```

```
scala> val (myVar1: Int, myVar2: String) = (10, "Hello Saif")
myVar1: Int = 10
myVar2: String = Hello Saif
```

Eclipse:

Method 1: Multiple Assignments by declaring Val with Data Type



```
1 package scala_practice
2
3 object Multiple_Assignments_Var {
4     def main (args: Array [String]) {
5         val (myVar1: Int, myVar2: String) = Pair (10, "Hello Saif")
6         println (myVar1)
7         println (myVar2)
8     }
9 }
```

<terminated> Multiple_Assignments_Var\$ [Scala Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.exe (24-Dec-2018, 10:28:01 PM)

10
Hello Saif

Method 2: Multiple Assignments by declaring Val without Data Type



```
1 package scala_practice
2
3 object Multiple_Assignments_Var {
4     def main (args: Array [String]) {
5         val (myVar1, myVar2) = Pair (10, "Hello Saif")
6         println (myVar1)
7         println (myVar2)
8     }
9 }
```

<terminated> Multiple_Assignments_Var\$ [Scala Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.exe (24-Dec-2018, 10:31:14 PM)

10
Hello Saif

String Interpolation:

String interpolation is a mechanism to combine your values inside a string with variables. By interpolating a string, we can embed variable references directly in a processed string literal.

The Scala notation for string interpolation is an “s” prefix added before the first double quote of the string. Then dollar sign operator (\$) (with optional braces) can be used to note references to external data.

s String Interpolator:

When we prepend ‘s’ to any string, we can directly use variables in it with the use of the ‘\$’ character. But that variable should be in scope of that string; it should be visible.

```
scala> val name = "Saif"
name: String = Saif

scala> println (s"Hello, $name! How are you?")
Hello, Saif! How are you?
```

See how simple that was? We can also process arbitrary expressions using \${}

```
scala> println (s"Do you think 2+3 is ${2+3}")
Do you think 2+3 is 5
```

f String Interpolator:

Let’s first see the problem with the s interpolator.

```
scala> val a = 77.0000
a: Double = 77.0

scala> println (s"Value of a is: $a")
Value of a is: 77.0
```

This print 77.0, not 77.0000

Now, see this with the f interpolator. The number is 77.0000

```
scala> println (f"Value of a is: $a%.4f")
Value of a is: 77.0000
```

This is quite like the printf style format specifiers in C. These are %d, %f, and so on.

```
scala> println(f"$name%s says the number is $a%.4f")
Saif says the number is 77.0000
```

This is **type-safe**. If the variable reference and the format specifier don’t match, it raises an error. By default, the specifier is %s.

raw String Interpolator:

In this String Interpolation, we should define String Literals with raw String Interpolator. That means String Literals should start with 'raw'.

It is used to accept escape sequence characters like '\n' as it is in a String literal. That means when we don't want to process or evaluate escape sequence characters then we should use raw String Interpolator.

How s String Interpolator Processes Escape Sequence Characters in a String Literal:

```
scala> val itemName = "Laptop"
itemName: String = Laptop

scala> val itemPrice = 50.00
itemPrice: Double = 50.0

scala> val str = s"Item Details: \nName: $itemName \t Price: $itemPrice"
str: String =
Item Details:
Name: Laptop      Price: 50.0
```

How raw String Interpolator works on Escape Sequence Characters in a String Literal:

```
scala> val itemName = "Laptop"
itemName: String = Laptop

scala> val itemPrice = 50.00
itemPrice: Double = 50.0

scala> val str = raw"Item Details: \nName: $itemName \t Price: $itemPrice"
str: String = Item Details: \nName: Laptop \t Price: 50.0
```

Scala Data Types:

| Data Type | Default Value | Size | Range |
|-----------|---------------|--|---|
| Boolean | False | true/false | NA |
| Byte | 0 | 8-bit signed value (-2 ⁷ to 2 ⁷ -1) | -128 to 127 |
| Short | 0 | 16-bit signed value (-2 ¹⁵ to 2 ¹⁵ -1) | -32768 to 32767 |
| Char | '\u0000' | 16-bit unsigned Unicode character (0 to 2 ¹⁶ -1) | U+0000 to U+FFFF |
| Int | 0 | 32-bit signed value (-2 ³¹ to 2 ³¹ -1) | - 2147483648 to 2147483647 |
| Long | 0L | 64-bit signed value (-2 ⁶³ to 2 ⁶³ -1) | -9223372036854775808 to 9223372036854775807 |
| Float | 0.0F | 32-bit IEEE 754 single-precision float | NA |
| Double | 0.0D | 64-bit IEEE 754 double-precision float | NA |
| String | Null | A sequence of characters | NA |

Escape Values:

An escape value is a backslash with a character that will escape that character to execute a certain functionality. We can use these in character and string literals. We have the following escape values in Scala:

| Escape Sequences | Unicode | Description |
|------------------|---------|--------------------|
| \b | \u0008 | Backspace BS |
| \t | \u0009 | Horizontal Tab HT |
| \n | \u000a | Newline |
| \f | \u000c | Formfeed FF |
| \r | \u000d | Carriage Return CR |
| \" | \u0022 | Double quote “ |
| \' | \u0027 | Single quote ‘ |
| \\ | \u005c | Backslash \ |

A character with Unicode ranging 0 and 255 may be represented by an octal escape, i.e., a backslash '\' which is further followed by a sequence of up to three octal characters.

Operator:

A symbol that instructs the compiler to perform certain mathematical or logical manipulations is an operator. We have the following types of operators in Scala:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators

Let's discuss all Scala operator one by one.

Arithmetic Operators:

This Scala operator tells Scala to perform arithmetic operations on two values. We have five of these.

```
scala> val a = 5  
a: Int = 5  
  
scala> val b = 2  
b: Int = 2
```

a) Addition (+):

This adds two operands.

```
scala> val c = a + b  
c: Int = 7
```

b) Subtraction (-):

This subtracts the second operand from the first.

```
scala> val c = a - b  
c: Int = 3
```

c) Multiplication (*)

This multiplies the two operands.

```
scala> val c = a * b  
c: Int = 10
```

d) Division (/):

This divides the first value by the second.

```
scala> val c = a / b  
c: Int = 2
```

e) Modulus (%):

This returns the remainder after dividing the first number by the second.

```
scala> val c = a % b  
c: Int = 1
```

Relational Operators:

Now, let's discuss the Scala operators that let us compare values. They are relational Scala Operators.

```
scala> val a = 5  
a: Int = 5
```

```
scala> val b = 2  
b: Int = 2
```

a) Equal to (==):

This returns true if the two values are equal; otherwise, false.

```
scala> val c = (a == b)  
c: Boolean = false
```

b) Not Equal to (!=):

This returns true if the two values are unequal; otherwise, false.

```
scala> val c = (a != b)  
c: Boolean = true
```

c) Greater Than (>):

This returns true if the first operand is greater than the second; otherwise, false.

```
scala> val c = a > b  
c: Boolean = true
```

d) Less Than (<):

This returns true if the first operand is lesser than the second; otherwise, false.

```
scala> val c = a < b  
c: Boolean = false
```

e) Greater Than or Equal to (>=):

This returns true if the first operand is greater than or equal to the second; otherwise, false.

```
scala> val c = (a >= b)
c: Boolean = true
```

f) Less Than or Equal to (<=):

This returns true if the first operand is less than or equal to the second; otherwise, false.

```
scala> val c = (a <= b)
c: Boolean = false
```

Logical Operators:

Moving on to logical Scala operators, we have three. Let's take Boolean values for this.

```
scala> val a = true
a: Boolean = true

scala> val b = false
b: Boolean = false
```

a) Logical AND (&&):

This returns true if both operands are true; otherwise, false.

```
scala> a && b
res1: Boolean = false
```

b) Logical OR (||):

This returns true if either or both operands are true; otherwise, false.

```
scala> a || b
res2: Boolean = true
```

c) Logical NOT (!):

This reverses the operand's logical state. It turns true to false, and false to true.

```
scala> !(a && b)
res3: Boolean = true
```

Assignment Operators:

Scala supports 11 different assignment operators:

a) Simple Assignment Operator (=):

This assigns the value on the right to the operand on the left.

```
scala> var a = 1 + 2  
a: Int = 3
```

b) Add and Assignment Operator (+=):

This Scala Operators adds the value of the right operand to the one on the left, and then assigns the result to the left operand.

```
scala> a = a + 2  
a: Int = 5  
  
scala> a += 2  
  
scala> a  
res35: Int = 7
```

This is the same as $a = a + 2$

c) Subtract and Assignment Operator (-=):

This operator in Scala subtracts the second operand from the first, then assigns the result to the left operand.

```
scala> a = a - 2  
a: Int = 5  
  
scala> a -= 3  
  
scala> a  
res37: Int = 2
```

This is the same as $a = a - 2$

d) Multiply and Assignment Operator (*=):

This Scala Operator multiplies the two operands, and then assigns the result to the left operand.

```
scala> var a = 2  
a: Int = 2  
  
scala> a = a * 2  
a: Int = 4  
  
scala> a *= 2  
  
scala> a  
res40: Int = 8
```

e) Divide and Assignment Operator (/=):

Divide and Assignment Scala Operator divides the first operand by the second, and then assigns the result to the left operand.

```
scala> var a = 2  
a: Int = 2  
  
scala> a /= 2  
  
scala> a  
res42: Int = 1
```

f) Modulus and Assignment Operator (%=):

Modulus and Assignment Scala Operator calculates the modulus remaining after dividing the first operand by the second. Then, it assigns this result to the first operand.

```
scala> var a = 2  
a: Int = 2  
  
scala> a %= 2  
  
scala> a  
res44: Int = 0
```

Conditional Expressions:

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

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

Scala if statement:

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

Syntax:

```
if(condition) {  
    // Statements to be executed  
}
```

```
scala> val age: Int = 20  
age: Int = 20  
  
scala> if{age > 18} {  
    | println("Age is greater than 18, i.e:" +age)  
    | }  
Age is greater than 18, i.e:20
```

Scala If-Else Statement:

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

Syntax:

```
if(condition) {  
    // If block statement to be executed  
} else {  
    // Else block statements to be executed  
}
```



```
scala> {  
  | val no: Int = 20  
  | if(no%2 == 0) {  
  | println("The Number: " +no +" is Even")  
  | } else {  
  | println("The Number: " +no +" is Odd")  
  | }  
  | }  
The Number: 20 is Even
```

```
scala> val no = 25  
no: Int = 25  
  
scala> println("The Number: " +no +" is Odd")  
The Number: 25 is Odd
```

Scala If-Else-If Ladder Statement:

An 'if' statement can be followed by an optional 'else if...else' statement, which is very useful to test various conditions using single if...else if statement.

When using if, else if, else statements there are few points to keep in mind.

- An 'if' can have zero or one else's and it must come after any else if's.
- An 'if' can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

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

Syntax:

```
if (condition1) {  
  //Code to be executed if condition1 is true  
} else if (condition2) {  
  //Code to be executed if condition2 is true  
} else if (condition3) {  
  //Code to be executed if condition3 is true  
}  
else {  
  //Code to be executed if all the conditions are false  
}
```

```
scala> {
  | val no: Int = 21
  | if(no%2 == 0) {
  |   println("The nos is Even")
  | } else if(no%3 == 0) {
  |   println("The nos is divisble by 3")
  | } else if(no%5 == 0) {
  |   println("The nos is divisble by 5")
  | } else {
  |   println("Not Sure")
  | }
  | }
The nos is divisble by 3
```

Scala Nested If Else Example:

You can put an 'if' or 'else if' statement inside another 'if' or 'else if' statement.

```
if (condition1) {
  //Code to execute if expression 1 is true
  if (condition2) {
    //Code to execute if expression 2 is true
  } else {
    //Code to execute if expression 2 is false
  }
}
else {
  //Code to execute if expression 1 is false
}
```

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val x=7
if(x<10)
{
  println("Saif")
  if(x<5)
  {
    println("Arif")
  }
  else
  {
    println("Trainer")
  }
}
else
{
  println("Developer!")
}

// Exiting paste mode, now interpreting.

Saif
Trainer
x: Int = 7
```

While Loop:

A Scala while loop first checks the condition. This may be any expression. Any non-zero value is true; zero is false.

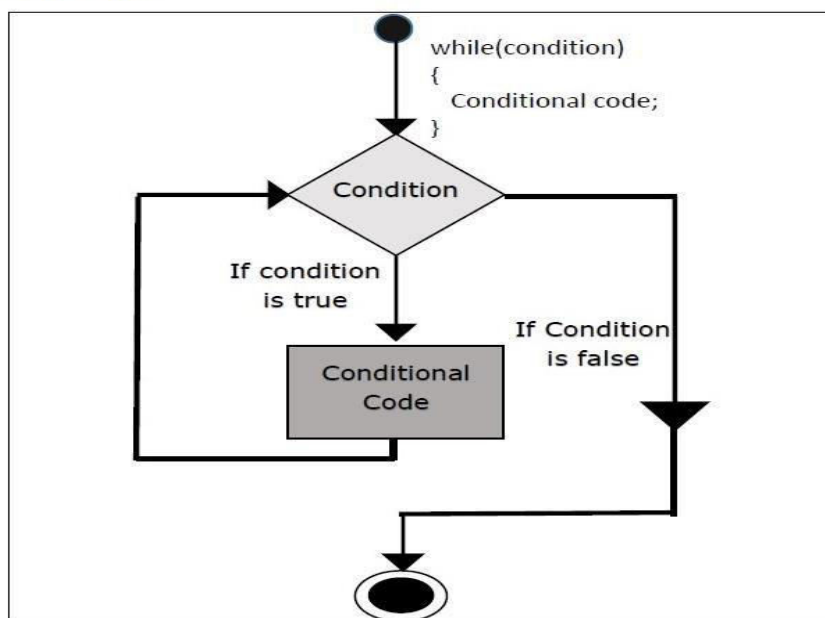
So, if the condition is true, it executes the code in the block under it. The block of code may be one statement or more. Then, it checks the condition again. If still true, it executes the block of code again. Otherwise, it skips to the first statement outside the loop.

If the condition is false the first time, the loop skips to the first statement outside the loop, instead.

Syntax:

```
while (condition) {
// Statement(s) to be executed
}
```

Flow Chart



```
package scala_practice

object Loop_While {
  var a = 2 // Initialization
  def main (args: Array [String]) {
    while (a <= 10) { // Condition
      println(a)
      a=a+2 // Incrementation
    }
  }
}
```

```
scala> {
  |   var x = 7
  |   while(x > 0) {
  |     println("Value of x is: " +x)
  |     x = x - 1
  |   }
  | }
Value of x is: 7
Value of x is: 6
Value of x is: 5
Value of x is: 4
Value of x is: 3
Value of x is: 2
Value of x is: 1
```

Initially, x is 7. The condition is true, because $7 > 0$. The loop reduces it to 6. And so on, this loop runs as long as x reaches 1. Then, inside the loop's body, it becomes 0. Now, when the condition is checked, it evaluates to false. So, it comes out of the loop.

Infinite While Loop Example:

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

It is easy to forget putting in the statement that modifies the variable involved in the condition. This creates an infinite loop- one that never ends, because the condition is always true.

```
package scala_practice

object Loop_While_Infinite {
  var x = 10
  def main (args: Array[String]) {
    while(true) {
      println (x)
    }
  }
}
```

```
scala> {
  |   var a=0
  |   while(a<7)
  |   {
  |     print(a)
  |   }
  | }_
```

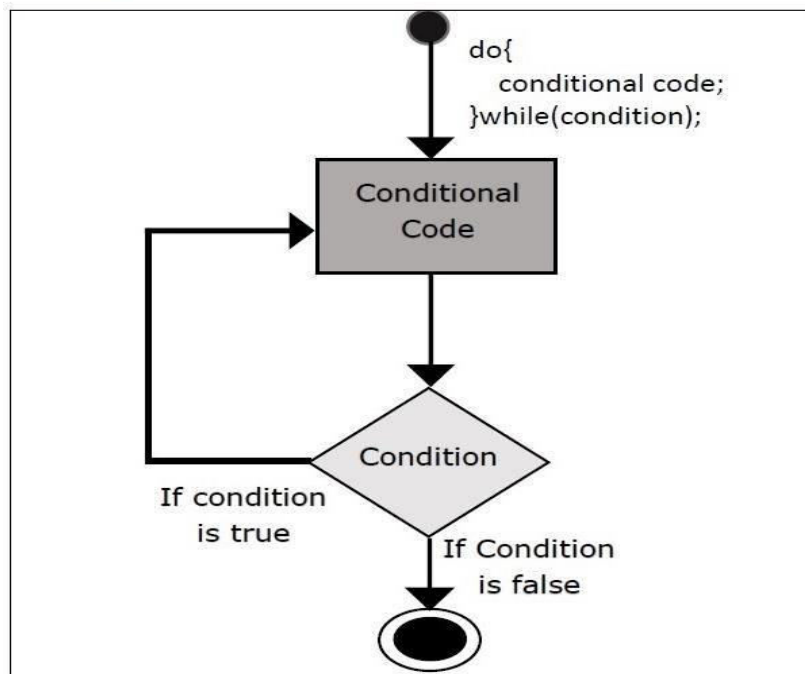
do while loop:

Unlike while loop, which tests the loop condition at the top of the loop, the do-while loop checks its condition at the bottom of the loop.

A Scala do while loop will execute a set of statements as long as a condition is true. This is like a while-loop. However, there are two differences:

- A while-loop checks the condition before entering the loop. A do-while loop checks it after executing the loop.
- This means that a do-while loop will execute at least once. But a while loop may never run.

Flow Chart



A Scala do while loop first executes the loop once. Then, it checks the condition, which may be any expression. While zero means false, any non-zero value is true.

The rest of the working is like a while-loop. If the condition is true, it executes the code block under itself. It can be one or more statements. Next, it checks the condition again. If true, it executes the block again. Otherwise, it skips to the first statement outside the do-while loop.

Even if the condition is false for the first time, the loop still runs at least once.

Syntax:

```
do
{
statement(s);
}
while(condition);
```

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

var x = 7
do
{
  println(x)
  x=x-1
}
while (x>0)

// Exiting paste mode, now interpreting.

7
6
5
4
3
2
1
x: Int = 0
```

Initially, x is 7. The loop executes once. This makes x equal to 6. Then, it checks the condition. Now since $6 > 0$, the condition is true. So, it executes the body yet again. This goes on until x becomes 1. The loop's body makes it 0. Then, it checks the condition, which is now false. Hence, it comes out of the loop.

Now, let's take the case when the condition is never true:

```
scala> {
  |     var x=0
  |     do
  |     {
  |         println(x)
  |         x=x-1
  |     }while(x>0)
  | }
0
```

```
package scala_practice
```

```
object Loop_Do_While {
  var a = 10;
  def main (args: Array[String]) {
    do {
      println ("Value of a: " + a);
      a = a + 1;
    }
    while (a < 20)
  }
}
```


Infinite do-while loop:

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

```
package scala_practice

object Loop_Infinite_Do_while {
  def main (args: Array[String]) {
    var a = 10
    do {
      println (a)
      a = a + 2;
    }
    while (true)
  }
}
```

Output:

10
12
14
16
?

Ctrl+Z

// To stop execution of program

for loop:

Scala for loop lets us execute specific code a certain number of times. A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

```
for (var x <- range) {
// statement(s) to be executed
}
```

Here, the Range could be a range of numbers and that is represented as i to j or sometime like i until j. The left-arrow ← operator is called a generator, so named because it's generating individual values from a range. In the above syntax, range is a value which has start and end point. You can pass range by using to or until keyword.

1) for loop by using to keyword / for i to j:

```
scala> {
|   for(a <- 1 to 5) {
|     println("Value of a is: " +a)
|   }
| }
Value of a is: 1
Value of a is: 2
Value of a is: 3
Value of a is: 4
Value of a is: 5
```

2) for loop by using until keyword / for i until j:

In the below example, until is used instead of to. The major difference between until and to is, to includes start and end value given in the range, while until excludes last value of the range. So, the below example will print only 1 to 4.

```
scala> {
|   for(a <- 1 until 5) {
|     println("Value of a is: " +a)
|   }
| }
Value of a is: 1
Value of a is: 2
Value of a is: 3
Value of a is: 4
```

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

3) Using Multiple Ranges in Scala for loop:

We can use multiple ranges in a Scala for loop. For this, we just separate the ranges with semicolons (;)

```
scala> {
  | for(a <- 1 to 3; b <- 1 until 4) {
  |   println("Value of a is: " +a)
  |   println("Value of b is: " +b)
  | }
  | }
Value of a is: 1
Value of b is: 1
Value of a is: 1
Value of b is: 2
Value of a is: 1
Value of b is: 3
Value of a is: 2
Value of b is: 1
Value of a is: 2
Value of b is: 2
Value of a is: 2
Value of b is: 3
Value of a is: 3
Value of b is: 1
Value of a is: 3
Value of b is: 2
Value of a is: 3
Value of b is: 3
```

4) for loop in Collection:

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

```
scala> {
  | val a: List[Int] = List(1,4,3,5)
  | for(i <- a) {
  |   println("Value of a is: " +a)
  | }
  | }
Value of a is: List(1, 4, 3, 5)
Value of a is: List(1, 4, 3, 5)
Value of a is: List(1, 4, 3, 5)
Value of a is: List(1, 4, 3, 5)
```

```
scala> {
  | val a = List(1,5,2,7)
  | for (i <- a) {
  |   val a = i
  |   println("Value of List Elements is: " +a)
  | }
  | }
Value of List Elements is: 1
Value of List Elements is: 5
Value of List Elements is: 2
Value of List Elements is: 7
```

5) for loop filtering Example:

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

```
scala> val a = List(1,3,5,7,9,11)
a: List[Int] = List(1, 3, 5, 7, 9, 11)
```

```
scala> for (i <- a
| if i>3; if i<11) {
| println(i)
| }
```

```
5
7
9
```

```
scala> for (i <- a; if i>3; if i<11) {
| println(i)
| }
```

```
5
7
9
```

```
scala> for (i <- a; if i>3 && i<11) {
| println(i)
| }
```

```
5
7
9
```

6) for loop by using yield keyword:

The last option is to return a value from a Scala for loop, and store it in a variable. We can also return it from a function. Then, we follow this by a yield statement. Note the curly braces.

```
scala> {
| val odd = List(1,3,5,7,9,11,13,15)
| val s = for(a <- odd; if a >3; if a < 13) yield a
| for(a <- s) {
| println(a)
| }
| }
```

```
5
7
9
11
```

```
scala> val x = List(1,3,5,7,9,11)
x: List[Int] = List(1, 3, 5, 7, 9, 11)

scala> var value = for {i <- x
    | if i > 3; if i < 11 } yield i
value: List[Int] = List(5, 7, 9)

scala> for (i <- value) {
    | println(i)
    | }
5
7
9
```

7) for each loop for Iterating Collection in 3 different types:

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

```
scala> val x = List(1,2,3,4)
x: List[Int] = List(1, 2, 3, 4)

scala> x.foreach {println}
1
2
3
4

scala> x.foreach (print)
1234
scala> x.foreach ((i:Int) => print(i+" "))
1 2 3 4
```

8) for loop using by keyword:

The by keyword is used to skip the iteration. When you code like: by 2 it means, this loop will add 2 in 1 and output is 3 and again add 2 output is 5 and so on...

```
scala> {
    | for(i <- 1 to 10 by 2) {
    |   println(i)
    | }
    | }
1
3
5
7
9
```

Scala Object Oriented Programming:

As we know, Scala is purely object-oriented. This means we can create classes and then objects from those classes. Now, let's see what classes and objects are.

A) Class:

A class is a blueprint for objects. Once you define a class, you can create objects from the class blueprint with the keyword **new**. Through the object you can use all functionalities of the defined class.

A class may hold the following members:

- Data members
- Member methods
- Constructors
- Blocks
- Nested classes
- Information about the superclass

In Scala, we must **initialize** all **instance variables**. Also, the **access scope** is **public** by **default**. We define the main method in an object. This is where the **execution** of the program begins.

B) Object:

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

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

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

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

An Example of Class & Object:

```
class Student {                                // Creating a class named as Student
    var id: Int = 101                          // All fields must be initialized
    var name: String = "Saif";
}

object MainObject {                          // Creating an object for execution
    def main (args: Array[String]) {
        var s = new Student ()               // Creating an object from class
        println (s.id+ " "+s.name);
    }
}
```

Output:

101 Saif


```

class Student(id: Int, name: String){           // Primary constructor
  def show(){
    println (id+ " "+name)
  }
}

object MainObject{
  def main(args: Array[String]){
    var s = new Student(100, "Saif")           // Passing values to constructor
    s.show()                                    // Calling a function by using an object
  }
}

```

```

5 Class_Object.scala
1 package scala_handson
2
3 // Creating class without Constructor
4 class Student {
5   val id: Int = 101
6   val name: String = "Saif"
7 }
8
9 // Creating class with Constructor
10 class Student1(id: Int, name: String) {
11   def show() {
12     println(id + " " + name)
13   }
14 }
15
16 object Class_Object {
17   // Main method to invoke the process
18   def main(args: Array[String]) {
19     // Creating an object from class without Constructor
20     val s = new Student()
21     println(s.id + " " + s.name)
22     // Creating an object from class with Constructor & passing the value as fields
23     val s1 = new Student1(102, "Ram")
24     s1.show()
25   }
26 }

```

Try a Program:

Create a class of Student with method getRecord and create object from the class to maintain the records of students.

Output:

```

101 Saif
102 Ram

```

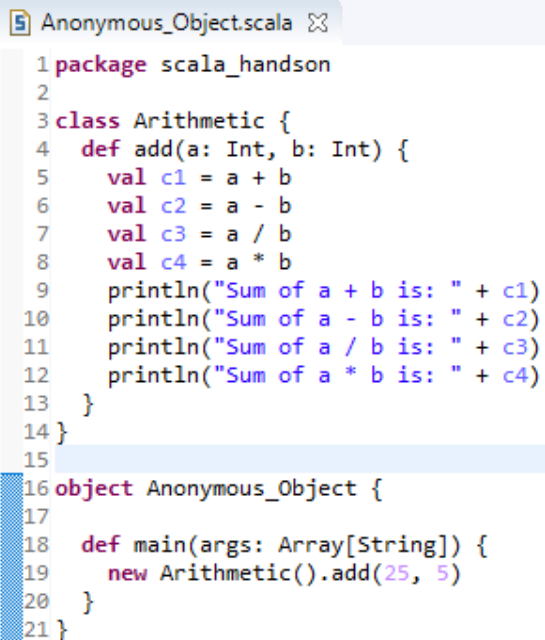
Scala Anonymous object:

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

```
package scala_handson
```

```
class Arithmetic {  
  def add(a: Int, b: Int) {  
    val c1 = a + b  
    val c2 = a - b  
    val c3 = a / b  
    val c4 = a * b  
    println("Sum of a + b is: " + c1)  
    println("Sum of a - b is: " + c2)  
    println("Sum of a / b is: " + c3)  
    println("Sum of a * b is: " + c4)  
  }  
}
```

```
object Anonymous_Object {  
  
  def main(args: Array[String]) {  
    new Arithmetic().add(25, 5)  
  }  
}
```

A screenshot of an IDE window titled 'Anonymous_Object.scala'. The code is as follows:

```
1 package scala_handson  
2  
3 class Arithmetic {  
4   def add(a: Int, b: Int) {  
5     val c1 = a + b  
6     val c2 = a - b  
7     val c3 = a / b  
8     val c4 = a * b  
9     println("Sum of a + b is: " + c1)  
10    println("Sum of a - b is: " + c2)  
11    println("Sum of a / b is: " + c3)  
12    println("Sum of a * b is: " + c4)  
13  }  
14 }  
15  
16 object Anonymous_Object {  
17  
18   def main(args: Array[String]) {  
19     new Arithmetic().add(25, 5)  
20   }  
21 }
```

Scala Singleton Object:

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

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

If you don't create singleton object, your code will compile successfully but will not produce any output. Methods declared inside Singleton Object are accessible globally. A singleton object can extend classes and traits.

```
package scala_handson

object SingletonObject {
  def hello() {
    println("Hello, This is Singleton Object")
  }
}

object Singleton {
  def main(args: Array[String]) {
    SingletonObject.hello() // No need to create object.
  }
}
```

```
SingletonObject.scala
1 package scala_handson
2
3 object SingletonObject {
4   def hello() {
5     println("Hello, This is Singleton Object")
6   }
7 }
8
9 object Singleton {
10   def main(args: Array[String]) {
11     SingletonObject.hello() // No need to create object.
12   }
13 }
```

Note:

- 1) Scala does not have static members, like Java. Instead, an object is used to hold members that span instances, such as constants.
- 2) Unlike classes, singleton objects cannot take parameter.

Scala Companion Object:

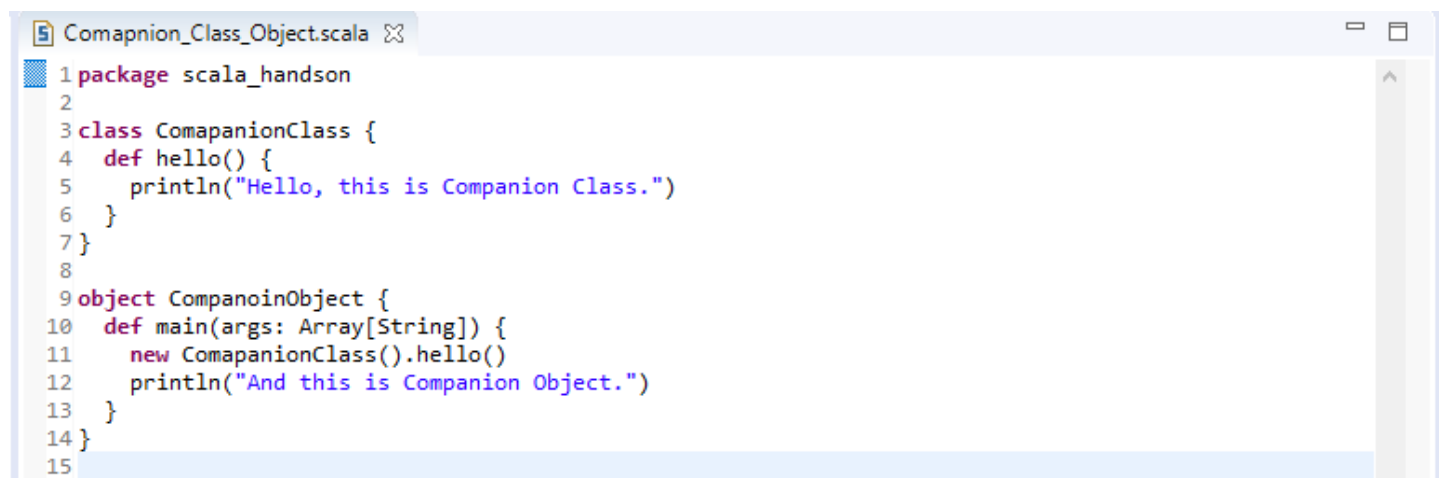
In scala, when you have a class with same name as singleton object, it is called companion class and the singleton object is called companion object.

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

```
package scala_handson

class CompanionClass {
  def hello() {
    println("Hello, this is Companion Class.")
  }
}

object CompanionObject {
  def main(args: Array[String]) {
    new CompanionClass().hello()
    println("And this is Companion Object.")
  }
}
```

A screenshot of a Scala IDE window titled 'Companion_Class_Object.scala'. The code is as follows:

```
1 package scala_handson
2
3 class CompanionClass {
4   def hello() {
5     println("Hello, this is Companion Class.")
6   }
7 }
8
9 object CompanionObject {
10  def main(args: Array[String]) {
11    new CompanionClass().hello()
12    println("And this is Companion Object.")
13  }
14 }
15
```

Scala Case Class:

A Scala Case Class is like a regular class, except it is good for modeling immutable data. It also serves useful in pattern matching; such a class has a default `apply()` method which handles object construction. A scala case class also has all **vals**, which means they are immutable.

Defining a Minimal Case Class in Scala:

To define a minimal Scala Case Class, we need the keywords 'case class' as an identifier and a parameter list. We can keep the parameter list empty.

```
scala> case class Song (title: String, artist: String, track: Int)
```

```
scala> case class Song (title: String, artist: String, track: Int)
defined class Song
```

Creating a Scala Object:

And now, it's time to create a Scala Object for this case class.

```
scala> val stay = Song ("Stay", "Inna", 4)
```

```
scala> val stay = Song ("Stay", "Inna", 4)
stay: Song = Song(Stay, Inna, 4)
```

Note: To create a Scala Object of a case class, we don't use the keyword 'new'. This is because its default `apply()` method handles the creation of objects.

Let's try accessing the title of this object:

```
scala> stay.artist
res1: String = Inna
```

```
scala> stay.track
res2: Int = 4
```

And now, let's try modifying it.

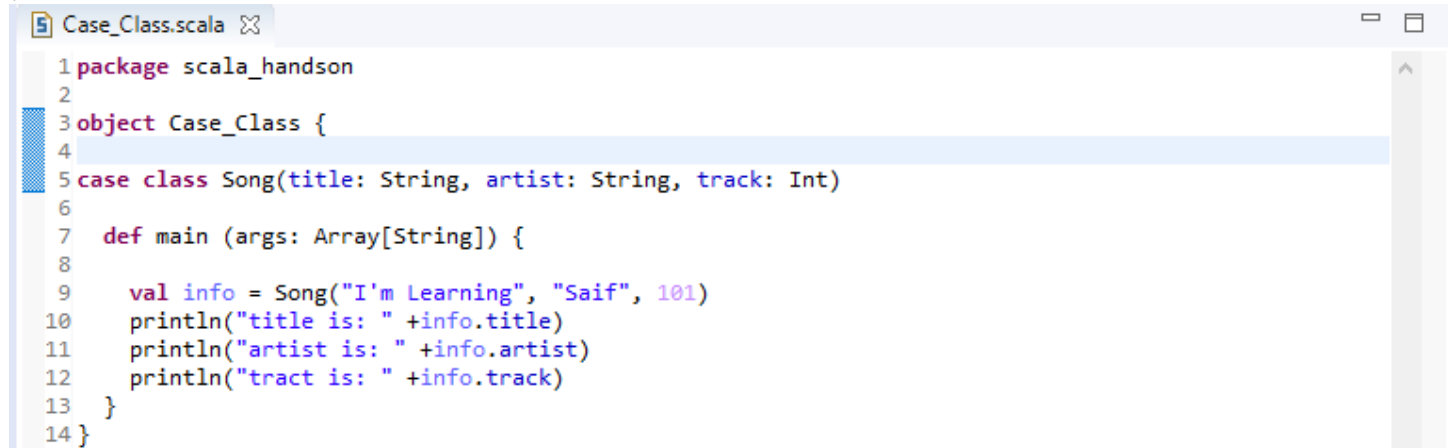
```
scala> stay.title = "Titanic"
<console>:10: error: reassignment to val
    stay.title = "Titanic"
           ^
```

```

package scala_handson

object Case_Class {
case class Song(title: String, artist: String, track: Int)
def main (args: Array[String]) {
    val info = Song("I'm Learning", "Saif", 101)
    println("title is: " + info.title)
    println("artist is: " + info.artist)
    println("track is: " + info.track)
}
}

```



```

1 package scala_handson
2
3 object Case_Class {
4
5 case class Song(title: String, artist: String, track: Int)
6
7 def main (args: Array[String]) {
8
9     val info = Song("I'm Learning", "Saif", 101)
10    println("title is: " + info.title)
11    println("artist is: " + info.artist)
12    println("track is: " + info.track)
13 }
14 }

```

Scala Pattern Matching:

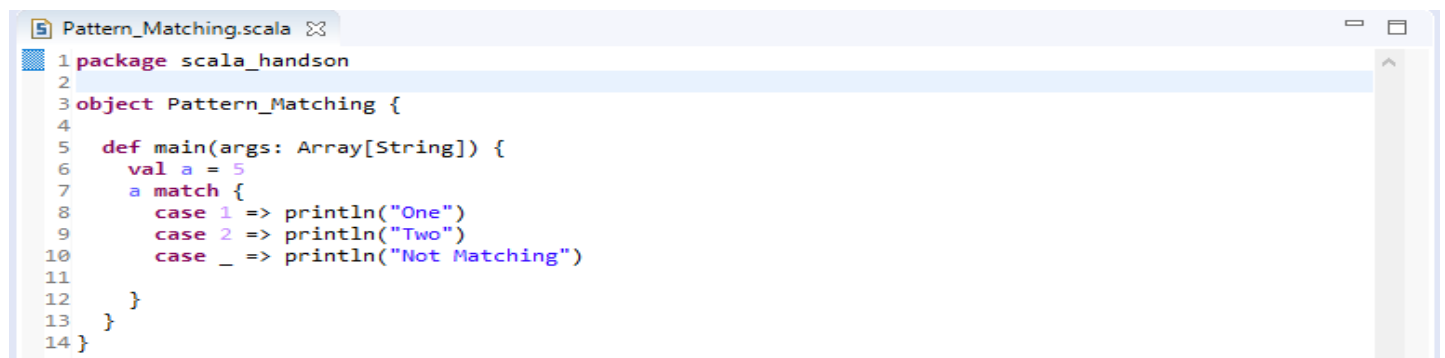
Pattern matching is a feature of scala. It works same as switch case in other programming languages. It matches best case available in the pattern.

```

package scala_handson

object Pattern_Matching {
def main(args: Array[String]) {
    val a = 5
    a match {
        case 1 => println("One")
        case 2 => println("Two")
        case _ => println("Not Matching")
    }
}
}

```



```

1 package scala_handson
2
3 object Pattern_Matching {
4
5 def main(args: Array[String]) {
6     val a = 5
7     a match {
8         case 1 => println("One")
9         case 2 => println("Two")
10        case _ => println("Not Matching")
11    }
12 }
13 }
14 }

```

Functions:

When we want to execute a group of statements to perform a task a multiple times, we don't type them again & again. We group them into a function, put a call to that function inside a loop, and make that loop run multiple times. Well, dividing our code into functions also makes for modularity: it makes it easier to debug and modify the code. Scala also has methods, but these differ slightly from Scala function.

Method: A method belongs to a class; it has a name, a signature [optionally] some annotations and some byte code.

Function: A function is a complete object that we can store in a variable. Then, a method is a function that is a member of some object.

Declaring a Function in Scala: To create a Scala function, we use the keyword 'def'.

Syntax:

```
def functionName ([list of parameters]): [return type] = {  
    function body  
    return [expr]  
}
```

Here, return type could be any valid Scala data type and list of parameters will be a list of variables separated by comma. Here list of parameters and return type are optional. Very similar to Java, a return statement can be used along with an expression in case function returns a value.

Following is the function which will add two integers and return their sum:

Syntax 1:

```
object add {  
    def addInt(a: Int, b: Int): Int = {  
        var sum: Int = 0  
        sum = a + b  
        return sum  
    }  
}
```

A function that does not return anything can return a **Unit** that is equivalent to **void** in Java and indicates that function does not return anything. The functions which do not return anything in Scala, they are called **procedures**.

Syntax 2:

```
object Hello {
  def printMe(): Unit = {
    println ("Hello, Scala!")
  }
}
```

Calling Function:

1) Scala provides a number of syntactic variations for invoking methods. Following is the standard way to call a method:

functionName (list of parameters)

2) If a function is being called using an instance of the object, then we would use dot notation similar to Java as follows:

[Instance].functionName (list of parameters)

E.g.

```
package scala_practice
```

```
object Function_Ex {
  def main (args: Array [String]) {
    println ("Returned Value: " + addInt (10, 20))
  }
  def addInt (a: Int, b: Int): Int = {
    var sum: Int = 0
    sum = a + b
    return sum
  }
}
```

The screenshot shows an IDE window titled 'Function_Ex.scala'. The code is as follows:

```
1 package scala_practice
2
3 object Function_Ex {
4   def addInt (a: Int, b: Int): Int = {
5     var sum: Int = 0
6     sum = a + b
7     return sum
8   }
9   def main (args: Array [String]) {
10    println ("Returned Value: " + addInt(10,20))
11  }
12 }
```

Below the code editor, the 'Console' tab is active, showing the output: `<terminated> Function_Ex$ [Scala Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.exe (31-Dec-2018, 10:50:30 PM) Returned Value: 30`

Default Parameter Values for a Function/Default Arguments:

Scala lets you specify default values for function parameters. The argument for such a parameter can optionally be omitted from a function call, in which case the corresponding argument will be filled in with the default. If you specify one of the parameters, then first argument will be passed using that parameter and second will be taken from default value.

```
scala> def product (a: Int, b: Int=7) {  
  | println {a*b}  
  | }  
product: (a: Int, b: Int)Unit
```

Let's try calling this with different number of arguments.

```
scala> product (7,0)  
0  
  
scala> product (7)  
49
```

Function with Variable Arguments:

When we don't know how many arguments we'll want to pass, we can use a variable argument for this:

```
scala> def sum (args: Int*): Int = {  
  | var result = 0  
  | for (arg <- args) {  
  |   result += arg  
  | }  
  | result  
  | }  
sum: (args: Int*)Int
```

Let's try calling this with different number of arguments.

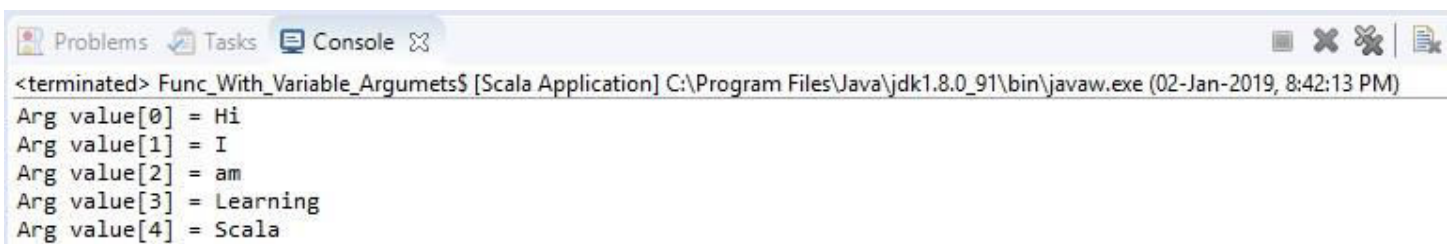
```
scala> sum (5)  
res0: Int = 5  
  
scala> sum (5 + 5)  
res1: Int = 10
```

Scala allows you to indicate that the last parameter to a function may be repeated. This allows clients to pass variable length argument lists to the function. Here, the type of args inside the print Strings function, which is declared as type "String*" is actually Array[String].

Try the following program, it is a simple example to show the function with arguments.

```
package scala_practice

object Func_With_Variable_Argumets {
  def printStrings (args: String*) = {
    var i: Int = 0
    for (s <- args) {
      println ("Arg value [" + i + "] = " + s)
      i = i + 1
    }
  }
  def main (args: Array[String]) {
    printStrings ("Hi", "I", "am", "Learning", "Scala")
  }
}
```



Nested Functions:

Scala allows you to define functions inside a function. Functions defined inside other functions are called local functions. Here is an implementation of a factorial calculator, where we use a conventional technique of calling a second, nested method to do the work.

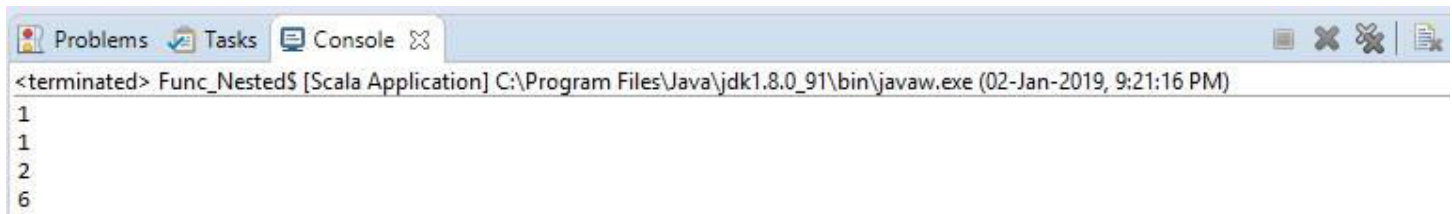
```
scala> def outer (a: Int) {
  |   println ("Outer Function Value is: " +a)
  |   def inner () {
  |     println (a * 3)
  |   }
  |   inner ()
  | }
outer: (a: Int)Unit
```

Let's try calling this with different number of arguments.

```
scala> outer (3)
Outer Function Value is: 3
9
```

```
package scala_practice

object Func_Nested {
  def factorial (i: Int): Int = {
    def fact (i: Int, res: Int): Int = {
      if (i < 1)
        res
      else
        fact (i - 1, i * res)
    }
    fact (i, 1)
  }
  def main (args: Array [String]) {
    println (factorial(0))
    println (factorial(1))
    println (factorial(2))
    println (factorial(3))
  }
}
```



The screenshot shows an IDE window with a tab labeled 'Console'. The title bar indicates the application is 'Func_Nested\$ [Scala Application]' running on 'C:\Program Files\Java\jdk1.8.0_91\bin\javaw.exe' at '02-Jan-2019, 9:21:16 PM'. The console output displays the results of the factorial function for inputs 0, 1, 2, and 3, which are 1, 1, 2, and 6 respectively.

```
<terminated> Func_Nested$ [Scala Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.exe (02-Jan-2019, 9:21:16 PM)
1
1
2
6
```

Functions with Named Arguments:

In a normal function call, the arguments in the call are matched one by one in the order of the parameters of the called function. Named arguments allow you to pass arguments to a function in a different order. The syntax is simply that each argument is preceded by a parameter name and an equals sign.

```
scala> def diff (a: Int, b: Int): Int = {
  |   return b - a
  | }
diff: (a: Int, b: Int)Int
```

Let's call function with different parameter!

```
scala> diff (2,5)
res0: Int = 3

scala> diff (b=2, a=5)
res1: Int = -3
```

```
package scala_handson
```

```
object function_named_arguments {
  def diff (a: Int, b: Int): Int = {
    b - a
  }
  def main (args: Array [String]) {
    val output = diff (2, 5)
    val output1 = diff (b=2, a=5)
    println (output)
    println (output1)
  }
}
```

Recursion Functions:

Recursion plays a big role in pure functional programming and Scala supports recursion functions very well. Recursion means a function can call itself repeatedly.

```
scala> def factorial (n: Int): Int = {
  | if (n == 1) { return 1 }
  | n * factorial (n - 1)
  | }
factorial: (n: Int)Int
```

Let's call function with different parameter!

```
scala> factorial(6)
res2: Int = 720

scala> factorial(3)
res3: Int = 6
```

```
package scala_practice
```

```
object Func_RecurSIONs {
  def factorial (n: Int): Int = {
    if (n <= 1)
      1
    else
      n * factorial (n - 1)
  }
  def main (args: Array [String]) {
    for (i <- 1 to 10)
      println ("Factorial of " + i + " = " + factorial(i))
  }
}
```

Output:

```
Factorial of 1 = 1
Factorial of 2 = 2
Factorial of 3 = 6
Factorial of 4 = 24
Factorial of 5 = 120
Factorial of 6 = 720
Factorial of 7 = 5040
Factorial of 8 = 40320
Factorial of 9 = 362880
Factorial of 10 = 3628800
```

Higher-Order Functions:

Scala allows the definition of higher-order functions. These are functions that take other functions as parameters, or whose result is a function.

```
package scala_handson

object function_higer_order {
  def addition (f: (Int, Int) => Int, a: Int, b: Int): Int = {
    f (a, b)
  }
  def main (args: Array [String]) {
    val squareSum = (x: Int, y: Int) => (x * x + y * y)
    val cubeSum = (x: Int, y: Int) => (x * x * x + y * y * y)
    val intSum = (x: Int, y: Int) => (x + y)

    val squaredSum = addition (squareSum, 1, 2)
    val cubedSum = addition (cubeSum, 1, 2)
    val normalSum = addition (intSum, 1, 2)

    println ("Square Output is: " + squaredSum)
    println ("Cube Output is: " + cubedSum)
    println ("Normal Sum Output is: " + normalSum)
  }
}
```

Output:

```
Square Output is: 5
Cube Output is: 9
Normal Sum Output is: 3
```

Anonymous Functions:

Scala provides a relatively lightweight syntax for defining anonymous functions. Anonymous functions in source code are called function literals and at run time, function literals are instantiated into objects called function values.

It is possible to define functions with multiple parameters as follows:

```
scala> val sum = (a: Int, b: Int) => a + b
sum: (Int, Int) => Int = <function2>
```

Variable sum is now a function that can be used the usual way:

```
scala> sum (5, 5)
res13: Int = 10

scala> println (sum(5, 5))
10
```

It is also possible to define functions with no parameter as follows

```
scala> val x = () => println ("Hello, Saif!")  
x: () => Unit = <function0>
```

Variable x is now a function that can be used the usual way:

```
scala> x()  
Hello, Saif!
```

```
package scala_handson
```

```
object function_anonymous {  
  def main (args: Array [String]) {  
    //Creating anonymous functions with multiple parameters  
    var myfunc1 = (str1: String, str2: String) => str1 + str2  
    //An anonymous function is created using _ wildcard instead of variable name  
    //because str1 and str2 variable appear only once  
    var myfunc2 = (_: String) + (_: String)  
    // Here, the variable invoke like a function call  
    println (myfunc1 ("Saif", "Shaikh"))  
    println (myfunc2 ("Teaching", "Scala"))  
  }  
}
```