Scalar programming language

Definition of Scala:

Scala is an abbreviation of the word Scalable Language. This indicates that Scala Programming language has the ability to grow with the growth in the technicalities of your program. The programmers can either go for playing with the language by typing expressions of one lines and going through the results or they may go for using this language for full scale technical programming missions.

The language is used for technical purposes by a wide range of companies, such as LinkedIn, Intel, or Twitter. Scala language consists of a very robust static system provides farfetched support for functional programming. The language is designed in a manner that makes it concise yet function. The pitfalls of Java were used as a basis for the development of a number of design decisions for Scala.

Key words of the scala:

Key words (or) reserved words are the words in a language that are used for some internal process or represent some predefined actions. These words are therefore not allowed to use as variable names or objects.

Example program:

object Main

{

    def main(args: Array[String])

    {

        var p = 10

        var q = 30

        var sum = p + q

        println("The sum of p and q is :"+sum);

    }

}

**Output:**

**The sum of p and q is:40**

class GFG

{

    var name = "Priyanka"

    var age = 20

    var branch = "Computer Science"

    def show()

    {

    println("Hello! my name is " + name + "and my age is"+age);

    println("My branch name is " + branch);

    }

}

Object main

{

    def main(args: Array[String])

    {

        var ob = new GFG();

        ob.show();

    }

}

Out put:

Hello! my name is Priyanka and my age is20

My branch name is Computer Science

# Scala identifiers:

Identifiers are used for identification purpose.in scala,an identifier can be a class name, method name,variable name or an object name.

Example:

class GFG{

var a: Int = 20

}

object Main {

def main(args: Array[String]) {

var ob = new GFG();

}

}

In the above program we have 6 identifiers:

* **GFG:**Class name
* **a:**Variable name
* **Main:**Object name
* **main:**Method name
* **args:**Variable name
* **ob:**Object name

#### Rules for defining Scala

There are certain rules for defining a valid Scala identifier. These rules must be followed; otherwise we get a **compile-time error**.

* Scala identifiers are **case-sensitive.**
* Scala does not allow you to use keyword as an identifier.
* **Reserved Words** can’t be used as an identifier like $ etc.
* Scala only allowed those identifiers which are created using below four types of identifiers.
* There is **no limit** on the length of the identifier, but it is advisable to use an optimum length of 4 – 15 letters only.
* Identifiers should **not** start with digits **([0-9])**. For example “123geeks” is a not a valid Scala identifier.

# Variables in scala:

Variables are simply storage locations. Every variable is known by its name and stores some known and unknown piece of information known as value. So one can define a variable by its data type and name, a data type is responsible for allocating memory for the variable. In [Scala](https://www.geeksforgeeks.org/introduction-to-scala/) there are two types of variables:

* Mutable Variables
* Immutable Variables

Let’s understand each one of these variables in detail.

**Mutable Variable:**   
These variables are those variables that allow us to change a value after the declaration of a variable. Mutable variables are defined by using the **var** keyword. The first letter of data type should be in capital letter because in Scala data type is treated as objects.   
**Syntax:**

var Variable name: Data type = "value";

**For Example:**

var name: String = "scala programming";

Here, *name*is the name of the variable, *string*is the data type of variable and *scala programming*is the value that store in the memory.   
*Another way of defining variable:*

**Syntax:**

var variable name = value

**For Example:**

*var value = 40*

*//it works without error   
value = 32*

 Here, the value is the name of the variable.

**Immutable Variable:**   
These variables are those variables that do not allow you to change a value after the declaration of a variable. Immutable variables are defined by using the **val** keyword. The first letter of data type should be in capital letter because in Scala data type is treated as objects.

**Syntax:**

val Variable name: Data type = "value";

**For Example:**

val name: String = “scala programming”

Here, a name is the name of the variable, a string is the data type of variable and scala programming is the value that store in the memory.   
Another way of defining variable:

**Syntax:**

val variable name = "value"

**For Example:**

*val value = 40*

*//it will give an error   
value = 32*

 Here value is the name of the variable.

**Rules for naming variable in Scala**

* Variable name should be in lower case.
* Variable name can contain letter, digit and two special characters(Underscore(\_) and Dollar($) sign)
* Variable name must not contain the keyword or reserved word.
* Starting letter of the variable name should be an alphabet.
* White space is not allowed in variable name.

**Note:** Scala supports multiple assignments but you can use multiple assignments only with immutable variables.

# Scala | Decision Making (if, if-else, Nested if-else, if-else if)

Decision making in programming is similar to decision making in real life. In decision making, a piece of code is executed when the given condition is fulfilled. Sometimes these are also termed as the Control flow statements. [Scala](https://www.geeksforgeeks.org/introduction-to-scala/" \t "_blank) uses control statements to control the flow of execution of the program based on certain conditions. These are used to cause the flow of execution to advance and branch based on changes to the state of a program.

**The conditional statements of Scala are:**

#### if

#### if-else

#### Nested if-else

#### if-else if ladder

#### if statement

“**if**” statement is the simplest decision making statements among all decision making statements. In this statement, the block of code is executed only when the given condition is true and if the condition is false then that block of code will not execute.

**Syntax:**

if(condition)

{

// Code to be executed

}

Here, **condition**after evaluation will be either true or false. if statement accepts boolean values – if the value is true then it will execute the block of statements under it.  
If we do not provide the curly braces ‘{‘ and ‘}’ after **if(condition)** then by default if statement will consider the immediate one statement to be inside

Its block.

**Example:**

if(condition)

statement1;

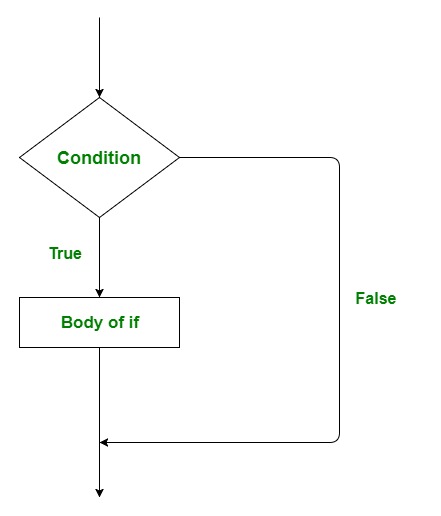
statement2;

// Here if the condition is true, if block

// will consider only statement1 to be inside

// its block.

**Flow Chart:**

[](https://media.geeksforgeeks.org/wp-content/uploads/if-statement-2.jpg)

**Example:**

object Test {  
 def main(args: Array[String]): Unit =  
 {  
 var a = 50  
 if(a>30)  
 *println*("the number is greater number")  
   
 }  
}

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| If-else statement The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won’t. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.  **Syntax:**  if (condition)  {  // Executes this block if  // condition is true  }  else  {  // Executes this block if  // condition is false  }  **Flow Chart:**  <https://media.geeksforgeeks.org/wp-content/uploads/if-else-statement.jpg>    **Example:**   |  | | --- | | object Test {  def main(args: Array[String]): Unit =  {  var a = 20  if(a>30)  *println*("the number is greater number")  else  *println*("the number is a smaller number")   } } |   **Output:**  the number is a smaller number Nested if-else statement A nested if is an if statement that is the target of another if-else statement. Nested if-else statement means an if-else statement inside an if statement or in a else statement. Scala allows us to nest if-else statements within if-else statement.  **Syntax:**  // Executes when condition\_1 is true  if (condition\_1)  {  if (condition\_2)  {  // Executes when condition\_2 is true  }  else  {  // Executes when condition\_2 is false  }  }  // Executes when condition\_1 is false  else  {    if (condition\_3)  {  // Executes when condition\_3 is true  }  else  {  // Executes when condition\_3 is false  }  }  **Flow Chart:**  <https://media.geeksforgeeks.org/wp-content/uploads/Nested-If-else.jpg>  **Example:**   |  | | --- | | // Scala program to illustrate  // the nested if-else statement  object Test {    // Main Method  def main(args: Array[String]) {        // taking three variables      var a: Int = 70      var b: Int = 40      var c: Int = 100        // condition\_1      if (a > b)      {          // condition\_2          if(a > c)          {              println("a is largest");          }            else          {              println("c is largest")          }        }        else      {             // condition\_3          if(b > c)          {              println("b is largest")          }            else          {              println("c is largest")          }      }  }  } |   **Output:**  c is largest if-else if Ladder Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.  **Syntax:**  if(condition\_1)  {  // this block will execute  // when condition\_1 is true  }  else if(condition\_2)  {  // this block will execute  // when condition2 is true  }  else  {  // this block will execute when none  // of the condition is true  }  **Flow Chart:**  <https://media.geeksforgeeks.org/wp-content/uploads/If-Else-if-Ladder.jpg>  **Example:**   |  | | --- | | object Test {  def main(args: Array[String]): Unit =  {  var a = 20  if(a>100)  *println*("the number is a greater ")  else if(a<100)  *println*("the number is smaller")   } } |   **Output:**  No Match Found  Looping in programming languages is a feature which facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true. Loops make the programmers task simpler. Scala provides the different types of loop to handle the condition based situation in the program. The loops in Scala are :   * [while Loop](https://www.geeksforgeeks.org/while-and-do-while-loop-in-scala/) * [do. While Loop](https://www.geeksforgeeks.org/while-and-do-while-loop-in-scala/) * [for Loop](https://www.geeksforgeeks.org/for-loop-in-scala/)   A while loop generally takes a condition in parenthesis. If the condition is **True**then the code within the body of the while loop is executed. A while loop is used when we don’t know the number of times we want the loop to be executed however we know the termination condition of the loop. It is also known as an*entry controlled loop* as the condition is checked before executing the loop. The while loop can be thought of as a repeating if statement. **Syntax:**   while (condition)  {  // Code to be executed  }  **Flowchart:**  https://media.geeksforgeeks.org/wp-content/uploads/while_loop.jpg     * While loop starts with the checking of the condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. For this reason, it is also called Entry control loop. * Once the condition is evaluated to true, the statements in the loop body are executed. Normally the statements contain an update value for the variable being processed for the next iteration. * When the condition becomes false, the loop terminates which marks the end of its life cycle.   **Example:**   * Scala  |  | | --- | | // Scala program to illustrate while loop  object whileLoopDemo  {  // Main method      def main(args: Array[String])      {   var x = 1;  / Exit when x becomes greater than 4          while (x <= 4)  {  println("Value of x: " + x);  // Increment the value of x for   // next iteration              x = x + 1;          }      }  } |   **Output:**   Value of x: 1  Value of x: 2  Value of x: 3  Value of x: 4  .  .    do. While Loop  A do. While loop is almost same as a while loop. The only difference is that do. While loop runs at least one time. The condition is checked after the first execution. A do. While loop is used when we want the loop to run at least one time. It is also known as the **exit controlled loop** as the condition is checked after executing the loop. **Syntax:**   do {  // statements to be Executed  } while(condition);  **Flowchart:**  https://media.geeksforgeeks.org/wp-content/uploads/do_while-Loop.jpg  **Example:**    * Scala  |  | | --- | | // Scala program to illustrate do..while loop  object dowhileLoopDemo  {  // Main method  def main(args: Array[String])      {  var a = 10  // using do..while loop          do          {              print(a + " ");              a = a - 1;          }while(a > 0);      }  } |   **Output:**  10 9 8 7 6 5 4 3 2 1    for Loop  *for* loop has similar functionality as while loop but with different syntax. for loops are preferred when the number of times loop statements are to be executed is known beforehand. There are many variations of “for loop in Scala” which we will discuss in upcoming articles. Basically, it is a repetition control structure which allows the programmer to write a loop that needs to execute a particular number of times. **Example:**    * Scala  |  | | --- | | // Scala program to illustrate for loop  object forloopDemo {       // Main Method     def main(args: Array[String]) {          var y = 0;          // for loop execution with range        for(y <- 1 to 7)        {           println("Value of y is: " + y);        }     }  } |   **Output:**   Value of y is: 1  Value of y is: 2  Value of y is: 3  Value of y is: 4  Value of y is: 5  Value of y is: 6  Value of y is: 7 Break statement in Scala In Scala, we use a break statement to break the execution of the loop in the program. Scala programing language does not contain any concept of break statement(in above 2.8 versions), instead of break statement, it provides a **break**method, which is used to break the execution of a program or a loop.  **Flow Chart:** https://media.geeksforgeeks.org/wp-content/uploads/Untitled-Diagram-35.jpg  **Syntax:**  // import package  import scala.util.control.\_  // create a Breaks object  val loop = new breaks;  // loop inside breakable  loop.breakable{  // Loop starts  for(..)  {  // code  loop.break  }  }  or  import scala.util.control.Breaks.\_  breakable  {  for(..)  {  code..  break  }  }  **For example:**   |  | | --- | | // Scala program to illustrate the  // implementation of break    // Importing break package  import scala.util.control.Breaks.\_  object MainObject  {    // Main method  def main(args: Array[String])  {        // Here, breakable is used to prevent exception      breakable      {          for (a <- 1 to 10)          {              if (a == 6)                    // terminate the loop when                  // the value of a is equal to 6                  break              else               println(a);          }      }  }  } |   **Output:**  1  2  3  4  5  **Break in Nested loop:** We can also use break method in nested loop.  **For example:**   |  | | --- | | // Scala program to illustrate the  // implementation of break in nested loop    // Importing break package  import scala.util.control.\_    object Test  {    // Main method  def main(args: Array[String])  {      var num1 = 0;      var num2 = 0;      val x = List(5, 10, 15);      val y = List(20, 25, 30);        val outloop = new Breaks;      val inloop = new Breaks;        // Here, breakable is used to      // prevent from exception      outloop.breakable      {          for (num1 <- x)          {                // print list x              println(" " + num1);                inloop.breakable              {              for (num2 <- y)              {                //print list y              println(" " + num2);                if (num2 == 25)              {                // inloop is break when              // num2 is equal to 25              inloop.break;              }              }            // Here, inloop breakable              }          }        // Here, outloop breakable      }  }  } |   **Output:**  5  20  25  10  20  25  15  20  25 Anonymous Functions in Scala: In Scala, An **anonymous function** is also known as a function literal. A function which does not contain a name is known as an anonymous function. An anonymous function provides a lightweight function definition. It is useful when we want to create an inline function. **Syntax:**  (z:Int, y:Int)=> z\*y  Or  (\_:Int)\*(\_Int)   * In the above first syntax, => is known as a transformer. The transformer is used to transform the parameter-list of the left-hand side of the symbol into a new result using the expression present on the right-hand side. * In the above second syntax, \_ character is known as a wildcard is a shorthand way to represent a parameter who appears only once in the anonymous function.   **Anonymous Functions With Parameters**  When a function literal is instantiated in an object is known as a function value. Or in other words, when an anonymous function is assigned to a variable then we can invoke that variable like a function call. We can define multiple arguments in the anonymous function.  **Example 1:**   |  | | --- | | // Scala program to illustrate the anonymous method  object Main  {      def main(args: Array[String])      {            // Creating anonymous functions          // with multiple parameters Assign          // anonymous functions to variables          var myfc1 = (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 myfc2 = (\_:String) + (\_:String)            // Here, the variable invoke like a function call          println(myfc1("Geeks", "12Geeks"))          println(myfc2("Geeks", "forGeeks"))      }  } |   **Output:**    Geeks12Geeks  GeeksforGeeks    **Anonymous Functions Without Parameters**  We are allowed to define an anonymous function without parameters. In Scala, We are allowed to pass an anonymous function as a parameter to another function. **Example 2:**   |  | | --- | | // Scala program to illustrate anonymous method  object Main  {      def main(args: Array[String])      {            // Creating anonymous functions          // without parameter          var myfun1 = () => {"Welcome to programming...!!"}          println(myfun1())            // A function which contain anonymous          // function as a parameter          def myfunction(fun:(String, String)=> String) =          {              fun("Dog", "Cat")          }            // Explicit type declaration of anonymous          // function in another function          val f1 = myfunction((str1: String,                      str2: String) => str1 + str2)            // Shorthand declaration using wildcard          val f2 = myfunction(\_ + \_)          println(f1)          println(f2)      }  } |   **Output:**  Welcome to programming…!!  DogCat  DogCat  Currying Functions in Scala with Examples  **Currying** in Scala is simply a technique or a process of transforming a function. This function takes multiple arguments into a function that takes single argument. It is applied widely in multiple functional languages.   **Syntax**  understand with a simple example, **Example:**   |  | | --- | | // Scala program add two numbers  // using currying Function  object Curry  {      // Define currying function      def add(x: Int, y: Int) = x + y;        def main(args: Array[String])      {          println(add(20, 19));      }  } |   **Output:**  39  Here, we have define **add** function which takes two arguments (x and y) and the function simply adds x and y and gives us the result, calling it in the main function.  **Another way to declare currying function** Suppose, we have to transform this add function into a Curried function, that is transforming the function that takes two(multiple) arguments into a function that takes one(single) argument.   **Syntax**    def function name(argument1) = (argument2) => operation  **Example**   |  | | --- | | // Scala program add two numbers  // using Currying function    object Curry  {      // transforming the function that      // takes two(multiple) arguments into      // a function that takes one(single) argument.      def add2(a: Int) = (b: Int) => a + b;        // Main method      def main(args: Array[String])      {          println(add2(20)(19));      }  } |   **Output:**  39  Here, we have define add2 function which takes only one argument **a** and we are going to return a second function which will have the value of add2. The second function will also take an argument let say **b** and this function when called in main, takes two parenthesis(add2()()), where the first parenthesis is of the function add2 and second parenthesis is of the second function. It will return the addition of two numbers, that is **a+b**. Therefore, we have curried the add function, which means we have transformed the function that takes two arguments into a function that takes one argument and the function itself returns the result.  **Currying Function Using Partial Application**  We have another way to use this Curried function and that is **Partially Applied function**. So, let’s take a simple example and understand. we have defined a variable sum in the main function  **Example**   |  | | --- | | // Scala program add two numbers  // using Currying function  object Curry  {      def add2(a: Int) = (b: Int) => a + b;        // Main function      def main(args: Array[String])      {          // Partially Applied function.          val sum = add2(29);          println(sum(5));      }  } |   **Output:**  34  Here, only one argument is passed while assigning the function to the value. The second argument is passed with the value and these arguments are added and result is printed.  Also, another way(syntax) to write the currying function.   **Syntax**  def function name(argument1) (argument2) = operation  **Example**   |  | | --- | | // Scala program add two numbers  // using Currying function  object Curry  {      // Curring function declaration      def add2(a: Int) (b: Int) = a + b;        def main(args: Array[String])      {          println(add2(29)(5));      }  } |   **Output:**  34  For this syntax, the Partial Application function also changes. **Example**   |  | | --- | | // Scala program add two numbers  // using Currying function  object Curry  {      // Curring function declaration      def add2(a: Int) (b: Int) = a + b;        def main(args: Array[String])      {         // Partially Applied function.          val sum=add2(29)\_;          println(sum(5));      }  } |   **Output:**  34  Here, only the **‘\_’** is added after the calling the function add2 for value of sum. Scala | Closures: **Scala Closures** are functions which uses one or more free variables and the return value of this function is dependent of these variable. The free variables are defined outside of the Closure Function and is not included as a parameter of this function. So the difference between a closure function and a normal function is the free variable. A **free variable** is any kind of variable which is not defined within the function and not passed as the parameter of the function. A free variable is not bound to a function with a valid value. The function does not contain any values for the free variable.  A closure function can further be classified into pure and impure functions, depending on the type of the free variable. If we give the free variable a type **var** then the variable tends to change the value any time throughout the entire code and thus may result in changing the value of the closure function. Thus this closure is a impure function. On the other-hand if we declare the free variable of the type **val** then the value of the variable remains constant and thus making the closure function a pure one.    **Example:**   |  | | --- | | // Addition of two numbers with  // Scala closure    // Creating object  object GFG  {      // Main method      def main(args: Array[String])      {          println( "Final\_Sum(1) value = " + sum(1))          println( "Final\_Sum(2) value = " + sum(2))          println( "Final\_Sum(3) value = " + sum(3))      }        var a = 4        // define closure function      val sum = (b:Int) => b + a  } |   **Output:**  Final\_Sum(1) value = 5  Final\_Sum(2) value = 6  Final\_Sum(3) value = 7  Here, In above program function **sum** is a closure function. var a = 4 is impure closure. the value of a is same and values of b is different. **Example:**   |  | | --- | | // Scala closure program to print a string  // Creating object  object GFG  {      // Main method      def main(args: Array[String])      {            var employee = 50            // define closure function          val gfg = (name: String) => println(s"Company name is $name"+                         s" and total no. of employees are $employee")            gfg("geeksforgeeks")      }  } |   **Output:**  Company name is geeksforgeeks and total no. of employees are 50. |

# Scala | Partially Applied functions

The **Partially applied functions** are the functions which are not applied on all the arguments defined by the stated function i.e, while invoking a function, we can supply some of the arguments and the left arguments are supplied when required. we call a function we can pass less arguments in it and when we pass less arguments it does not throw an exception. these arguments which are not passed to function we use hyphen( \_ ) as placeholder.  
**Some important points:**

* Partially applied functions are helpful in minimizing a function which accepts many arguments to a function that accepts only some arguments.
* It can replace any number of arguments defined by a function.
* It is easier for users to utilize this method.

**Syntax:**

val multiply = (a: Int, b: Int, c: Int) => a \* b \* c

// less arguments passed

val f = multiply(1, 2, \_: Int)

As we can see in above syntax we defined a normal function multiply which have three arguments we pass less arguments (two). we can see it does not throw an exception that is partially applied function.  
**Example:**

|  |
| --- |
| // Scala program of Partially  // applied functions    // Creating object  object Applied  {        // Main method      def main(args: Array[String])      {            // Creating a Partially applied          // function          def Book(discount: Double, costprice: Double)          : Double =          {                (1 - discount/100) \* costprice            }            // Applying only one argument          val discountedPrice = Book(25, \_: Double)            // Displays discounted price          // of the book          println(discountedPrice(400))        }  } |

**Output:**

300.0

Here, the *discount* is passed in the argument and *costprice* part is left empty which can be passed later when required so, the discounted price can be calculated any number of times.

**Some more examples of Partially applied functions:**

1. A partially applied function can be obtained even when not applied on any of the arguments defined.  
   **Example:**

|  |
| --- |
| // Scala program of Partially  // applied functions    // Creating object  object Applied  {        // Main method      def main(args: Array[String])      {            // Creating a Partially applied          // function          def Mul(x: Double, y: Double)          : Double =          {              x \* y          }            // Not applying any argument          val r = Mul \_            // Displays Multiplication          println(r(9, 8))      }  } |

1. **Output:**
2. 72.0
3. Partially applied functions can be utilized to replace any number of parameters.  
   **Example:**

|  |
| --- |
| // Scala program of Partially  // applied functions    // Creating object  object Applied  {        // Main method      def main(args: Array[String])      {            // Creating a Partially applied          // function          def Mul(x: Double, y: Double, z: Double)          : Double =          {              x \* y \* z          }            // applying some arguments          val r = Mul(7, 8, \_ : Double)            // Displays Multiplication          println(r(10))      }  } |

1. **Output:**
2. 560.0
3. *Currying* approach can be utilized in Partially applied functions to transmit a function with multiple arguments into multiple functions, where each function takes only one argument.  
   **Example:**

|  |
| --- |
| // Scala program of Partially  // applied functions using  // Currying approach    // Creating object  object curr  {        // Main method      def main(args: Array[String])      {            // Creating a Partially applied          // function          def div(x: Double, y: Double)          : Double =          {              x / y          }            // applying currying approach          val m = (div \_).curried            // Displays division          println(m(72)(9))      }  } |

1. **Output:**
2. 8.0
3. Here, currying approach breaks the given function into two functions, where each function takes one parameter so, you need to pass one value for each of the functions and get the desired output.

# Higher Order Functions in Scala:

A function is called **Higher Order Function** if it contains other functions as a parameter or returns a function as an output i.e, the functions that operate with another functions are known as Higher order Functions. It is worth knowing that this higher order function is applicable for functions and methods as well that takes functions as parameter or returns a function as a result. This is practicable as the compiler of Scala allows to force methods into functions.  
**Some important points about Higher order functions:**

* The Higher order functions are possible, as Scala programming language acts towards the functions as first-class values, which implies that analogous to some other values, functions can even be passed as a parameter or can be returned as an output, which is helpful in supplying an adjustable method for writing codes.
* It is beneficial in producing function composition where, functions might be formed from another functions. The function composition is the method of composing where a function shows the utilization of two composed functions.
* It is also constructive in creating lambda functions or anonymous functions. The [anonymous functions](https://www.geeksforgeeks.org/anonymous-functions-in-scala/) are the functions which does not has name, though perform like a function.
* It is even utilized in minimizing redundant lines of code from a program.

Now, lets see some examples.

* **Example :**

|  |
| --- |
| // Scala program of higher order  // function    // Creating object  object GfG  {        // Main method      def main(args: Array[String])      {          // Displays output by assigning          // value and calling functions          println(apply(format, 32))        }        // A higher order function      def apply(x: Double => String, y: Double) = x(y)        // Defining a function for      // the format and using a      // method toString()      def format[R](z: R) = "{" + z.toString() + "}"    } |

* **Output:**
* {32.0}
* Here, the apply function contains an another function **x** with a value **y** and applies the function x to y.
* **Example :**

|  |
| --- |
| // Scala program of higher order  // function    // Creating object  object GfG  {      // Main method      def main(args: Array[String])      {            // Creating a list of numbers          val num = List(7, 8, 9)            // Defining a method          def multiplyValue = (y: Int) => y \* 3            // Creating a higher order function          // that is assigned to the variable          val result = num.map(y => multiplyValue(y))            // Displays output          println("Multiplied List is: " + result)      }  } |

* **Output:**
* Multiplied List is: List(21, 24, 27)

# Currying Functions in Scala with Examples

**Currying** in Scala is simply a technique or a process of transforming a function. This function takes multiple arguments into a function that takes single argument. It is applied widely in multiple functional languages.  
   
**Syntax**

def function name(argument1, argument2) = operation

Let’s understand with a simple example,  
**Example:**

|  |
| --- |
| // Scala program add two numbers  // using currying Function  object Curry  {      // Define currying function      def add(x: Int, y: Int) = x + y;        def main(args: Array[String])      {          println(add(20, 19));      }  } |

**Output:**

39

Here, we have define **add** function which takes two arguments (x and y) and the function simply adds x and y and gives us the result, calling it in the main function.

**Another way to declare currying function**  
Suppose, we have to transform this add function into a Curried function, that is transforming the function that takes two(multiple) arguments into a function that takes one(single) argument.  
   
**Syntax**

def function name(argument1) = (argument2) => operation

**Example**

|  |
| --- |
| // Scala program add two numbers  // using Currying function    object Curry  {      // transforming the function that      // takes two(multiple) arguments into      // a function that takes one(single) argument.      def add2(a: Int) = (b: Int) => a + b;        // Main method      def main(args: Array[String])      {          println(add2(20)(19));      }  } |

**Output:**

39

Here, we have define add2 function which takes only one argument **a** and we are going to return a second function which will have the value of add2. The second function will also take an argument let say **b** and this function when called in main, takes two parenthesis(add2()()), where the first parenthesis is of the function add2 and second parenthesis is of the second function. It will return the addition of two numbers, that is **a+b**. Therefore, we have curried the add function, which means we have transformed the function that takes two arguments into a function that takes one argument and the function itself returns the result.

**Currying Function Using Partial Application:**

We have another way to use this Curried function and that is **Partially Applied function**. So, let’s take a simple example and understand. we have defined a variable sum in the main function

**Example**

|  |
| --- |
| // Scala program add two numbers  // using Currying function  object Curry  {      def add2(a: Int) = (b: Int) => a + b;        // Main function      def main(args: Array[String])      {          // Partially Applied function.          val sum = add2(29);          println(sum(5));      }  } |

**Output:**

34

Here, only one argument is passed while assigning the function to the value. The second argument is passed with the value and these arguments are added and result is printed.

Also, another way(syntax) to write the currying function.  
   
**Syntax**

def function name(argument1) (argument2) = operation

**Example**

|  |
| --- |
| // Scala program add two numbers  // using Currying function  object Curry  {      // Curring function declaration      def add2(a: Int) (b: Int) = a + b;        def main(args: Array[String])      {          println(add2(29)(5));      }  } |

**Output:**

34

For this syntax, the Partial Application function also changes.  
**Example**

|  |
| --- |
| // Scala program add two numbers  // using Currying function  object Curry  {      // Curring function declaration      def add2(a: Int) (b: Int) = a + b;        def main(args: Array[String])      {         // Partially Applied function.          val sum=add2(29)\_;          println(sum(5));      }  } |

**Output:**

34

Here, only the **‘\_’** is added after the calling the function add2 for value of sum.

# Scala | String Interpolation

**String Interpolation** refers to substitution of defined variables or expressions in a given String with respected values. String Interpolation provides an easy way to process String literals. To apply this feature of Scala, we must follow few rules:

1. String must be defined with starting character as **s** / **f**/**raw**.
2. Variables in the String must have ‘$’ as prefix.
3. Expressions must be enclosed within curly braces ({, }) and ‘$’ is added as prefix.

**Syntax:**

// x and y are defined

val str = s"Sum of $x and $y is ${x+y}"

**Types of String Interpolator**

1. **s Interpolator:** Within the String, we can access variables, object fields, functions calls, etc.

**Example 1:** variables and expressions:

|  |
| --- |
| // Scala program  // for s interpolator    // Creating object  object GFG  {      // Main method      def main(args:Array[String])      {       val x = 20          val y = 10          // without s interpolator          val str1 = "Sum of $x and $y is ${x+y}"            // with s interpolator          val str2 = s"Sum of $x and $y is ${x+y}"          println("str1: "+str1)          println("str2: "+str2)      }  } |

**Output:**

str1: Sum of $x and $y is ${x+y}

str2: Sum of 20 and 10 is 30

**Example 2:** function call

|  |
| --- |
| // Scala program  // for s interpolator   // Creating object  object GFG  {      // adding two numbers      def add(a:Int, b:Int):Int      =      {          a+b      }        // Main method      def main(args:Array[String])      {           val x = 20          val y = 10             // without s interpolator          val str1 = "Sum of $x and $y is ${add(x, y)}"      // with s interpolator          val str2 = s"Sum of $x and $y is ${add(x, y)}"           println("str1: " + str1)          println("str2: " + str2)      }  } |

**Output:**

str1: Sum of $x and $y is ${add(x, y)}

str2: Sum of 20 and 10 is 30

1. **f Interpolator:** This interpolation helps in formatting numbers easily.

To understand how format specifiers work refer [Format Specifiers](https://www.geeksforgeeks.org/format-specifiers-in-c/).

**Example 1:** printing upto 2 decimal place:

|  |
| --- |
| // Scala program  // for f interpolator   // Creating object  object GFG  {      // Main method      def main(args:Array[String])      {   val x = 20.6    // without f interpolator       val str1 = "Value of x is $x%.2f"     // with f interpolator        val str2 = f"Value of x is $x%.2f"           println("str1: " + str1)          println("str2: " + str2)         }  } |

**Output:**

str1: Value of x is $x%.2f

str2: Value of x is 20.60

**Example 2:** setting width in integers:

|  |
| --- |
| // Scala program  // for f interpolator   // Creating object  object GFG  {      // Main method      def main(args:Array[String])      {  val x = 11      // without f interpolator          val str1 = "Value of x is $x%04d"     // with f interpolator          val str2 = f"Value of x is $x%04d"        println(str1)          println(str2)      }  } |

**Output:**

Value of x is $x%04d

Value of x is 0011

If we try to pass a **Double**value while formatting is done using **%d** specifier, compiler outputs an error. In case of **%f** specifier, passing **Int** is acceptable.

1. **raw Interpolator:** String Literal should start with ‘raw’. This interpolator treats escape sequences same as any other character in a String.  
   **Example :**printing escape sequence:

|  |
| --- |
| // Scala program  // for raw interpolator   // Creating object  object GFG  {      // Main method      def main(args:Array[String])      {           // without raw interpolator          val str1 = "Hello\nWorld"   // with raw interpolator          val str2 = raw"Hello\nWorld”     println("str1: " + str1)          println("str2: " + str2)      }  } |

1. **Output:**
2. str1: Hello
3. World
4. str2: Hello\nWorld

**PATTERN MATCHING**

Pattern matching is the second most widely used feature of Scala, after function values and closures. Scala provides great support for pattern matching, in processing the messages.

A pattern match includes a sequence of alternatives, each starting with the keyword **case**. Each alternative includes a **pattern** and one or more **expressions**, which will be evaluated if the pattern matches. An arrow symbol => separates the pattern from the expressions.

Try the following example program, which shows how to match against an integer value.

object Demo {

def main(args: Array[String])

{

println(matchTest("two"))

println(matchTest("test"))

println(matchTest(1))

}

def matchTest(x: Any): Any = x match

{

case 1 => "one"

case "two" => 2

case y: Int => "scala.Int"

case \_ => "many"

}

**ARRAYS IN SCALA**

Scala provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

**Declaring an array**

To use an array in a program, you must declare a variable to reference the array and you must specify the type of array the variable can reference.

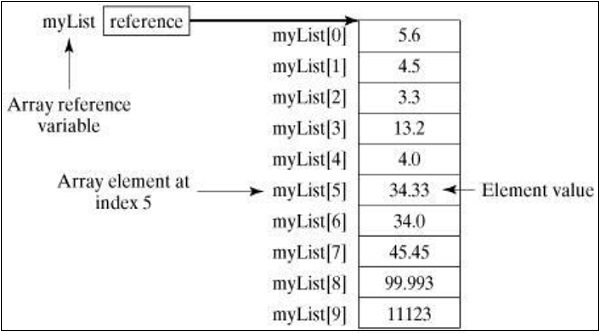
The following is the syntax for declaring an array variable.

### Syntax

var z:Array[String] = new Array[String](3)

Or

var z = new Array[String](3)



object Demo {

def main(args: Array[String]) {

var myList = Array(1.9, 2.9, 3.4, 3.5)

// Print all the array elements

for ( x <- myList ) {

println( x ) }

// Summing all elements

var total = 0.0;

for ( i <- 0 to (myList.length - 1)) {

total += myList(i);

}

println("Total is " + total);

// Finding the largest element

var max = myList(0);

for ( i <- 1 to (myList.length - 1) ) {

if (myList(i) > max) max = myList(i);

}

println("Max is " + max);

}

}

**SCALA LISTS**

A **list** is a collection which contains immutable data. List represents linked list in Scala. The **Scala List** class holds a sequenced, linear list of items.  
Following are the point of difference between lists and array in Scala:

* Lists are immutable whereas arrays are mutable in Scala.
* Lists represents a linked list whereas arrays are flat.

**Syntax :**

val variable\_name: List[type] = List(item1, item2, item3)

Or

val variable\_name = List(item1, item2, item3)

**Some important points about list in Scala:**

* In a Scala list, each element must be of the same type.
* The implementation of lists uses mutable state internally during construction.
* In Scala, list is defined under scala.collection.immutable package.
* A List has various methods to add, prepend, max, min, etc. to enhance the usage of list.

**Example 1:**

// Scala program to print immutable lists

import scala.collection.immutable.\_

// Creating object

object GFG

{

// Main method

def main(args:Array[String])

{

// Creating and initializing immutable lists

val mylist1: List[String] = List("Geeks", "GFG",

"GeeksforGeeks", "Geek123")

val mylist2 = List("C", "C#", "Java", "Scala",

"PHP", "Ruby")

// Display the value of mylist1

println("List 1:")

println(mylist1)

// Display the value of mylist2 using for loop

println("\nList 2:")

for(mylist<-mylist2)

{

println(mylist)

}

}

}

Output: List 1:List(Geeks, GFG, GeeksforGeeks, Geek123)

List 2:

C

C#

Java

Scala

PHP

Ruby

**Example2 :**

// Scala program to illustrate the

// use of empty list

import scala.collection.immutable.\_

// Creating object

object GFG

{

// Main method

def main(args:Array[String])

{

// Creating an Empty List.

val emptylist: List[Nothing] = List()

println("The empty list is:")

println(emptylist)

}

}

Output:

The empty list is:

List()

**Basic Operations on Lists**

The following are the three basic operations which can be performed on list in scala:

1. **head:** The first element of a list returned by head method.  
   Syntax: list.head //returns head of the list

**2.tail:** This method returns a list consisting of all elements except the first.  
Syntax: list.tail //returns a list consisting of all elements except the first

**3.isEmpty:** This method returns true if the list is empty otherwise false.  
Syntax: list.isEmpty //returns true if the list is empty otherwise false.

// Scala program of a list to

// perform head operation

import scala.collection.immutable.\_

// Creating object

object GFG

{

// Main method

def main(args:Array[String])

{

// Creating a List.

val mylist = List("C", "C#", "Java", "Scala",

"PHP", "Ruby")

println("The head of the list is:")

println(mylist.head)

println("The tail of the list is:")

println(mylist.tail)

println("List is empty or not:")

println(mylist.isEmpty)

}

}

Output: The head of the list is:

C

The tail of the list is:

List(C#, Java, Scala, PHP, Ruby)

List is empty or not:

False

## **Concatenating Lists**

You can use either **:::** operator or **List.:::()** method or **List.concat()** method to add two or more lists. Please find the following example given below −

### Example

object Demo {

def main(args: Array[String]) {

val fruit1 = "apples" :: ("oranges" :: ("pears" :: Nil))

val fruit2 = "mangoes" :: ("banana" :: Nil)

// use two or more lists with ::: operator

var fruit = fruit1 ::: fruit2

println( "fruit1 ::: fruit2 : " + fruit )

// use two lists with Set.:::() method

fruit = fruit1.:::(fruit2)

println( "fruit1.:::(fruit2) : " + fruit )

// pass two or more lists as arguments

fruit = List.concat(fruit1, fruit2)

println( "List.concat(fruit1, fruit2) : " + fruit )

}

}

Output:

fruit1 ::: fruit2 : List(apples, oranges, pears, mangoes, banana)

fruit1.:::(fruit2) : List(mangoes, banana, apples, oranges, pears)

List.concat(fruit1, fruit2) : List(apples, oranges, pears, mangoes, banana)

**FUNCTIONAL PROGRAMMING**

Scala lets you write code in an object-oriented programming (OOP) style, a functional programming (FP) style, and even in a hybrid style, using both approaches in combination.

# **PURE FUNCTIONS**

A first feature Scala offers to help you write functional code is the ability to write pure functions. In [Functional Programming, Simplified](https://alvinalexander.com/scala/functional-programming-simplified-book), Alvin Alexander defines a *pure function* like this:

* The function’s output depends *only* on its input variables
* It doesn’t mutate any hidden state
* It doesn’t have any “back doors”: It doesn’t read data from the outside world (including the console, web services, databases, files, etc.), or write data to the outside world.

As a result of this definition, any time you call a pure function with the same input value(s), you’ll always get the same result. For example, you can call a double function an infinite number of times with the input value 2, and you’ll always get the result 4.

## **Examples of pure functions**

Given that definition of pure functions, as you might imagine, methods like these in the *scala.math.\_* package are pure functions:

* abs
* ceil
* max
* min

These Scala String methods are also pure functions:

* is Empty
* length
* substring

Many methods on the Scala collections classes also work as pure functions, including drop, filter, and map.

## **Examples of impure functions**

Date and time related methods like getDayOfWeek, getHour, and getMinute are all impure because their output depends on something other than their input parameters. Their results rely on some form of hidden I/O, *hidden input* in these examples.

In general, impure functions do one or more of these things:

* Read hidden inputs, i.e., they access variables and data not explicitly passed into the function as input parameters
* Write hidden outputs
* Mutate the parameters they are given
* Perform some sort of I/O with the outside world

## **Writing pure functions**

Writing pure functions in Scala is one of the simpler parts about functional programming: You just write pure functions using Scala’s method syntax. Here’s a pure function that doubles the input value it’s given:

**def** **double**(i: **Int**): **Int** = i \* 2

Although recursion isn’t covered in this book, if you like a good “challenge” example, here’s a pure function that calculates the sum of a list of integers (List[Int]):

**def** **sum**(list: **List**[**Int**]): **Int** = list **match** {

**case** **Nil** => 0 **case** head :: tail => head + sum(tail)

}

**SCALA –SETS**

Scala Set is a collection of pairwise different elements of the same type. In other words, a Set is a collection that contains no duplicate elements. There are two kinds of Sets, the **immutable** and the **mutable**. The difference between mutable and immutable objects is that when an object is immutable, the object itself can't be changed.

By default, Scala uses the immutable Set. If you want to use the mutable Set, you'll have to import **scala.collection.mutable.Set** class explicitly. If you want to use both mutable and immutable sets in the same collection, then you can continue to refer to the immutable Set as **Set** but you can refer to the mutable Set as **mutable.Set**.

Here is how you can declare immutable Sets −

// Empty set of integer type

var s : Set[Int] = Set()

// Set of integer type

var s : Set[Int] = Set(1,3,5,7)

or

var s = Set(1,3,5,7)

While defining an empty set, the type annotation is necessary as the system needs to assign a concrete type to variable.

## **Basic Operations on set**

All operations on sets can be expressed in terms of the following three methods −

|  |  |
| --- | --- |
| **Sr.No** | **Methods & Description** |
| 1 | **head**  This method returns the first element of a set. |
| 2 | **tail**  This method returns a set consisting of all elements except the first. |
| 3 | **isEmpty**  This method returns true if the set is empty otherwise false. |

Try the following example showing usage of the basic operational methods −

### Example

object Demo

{

def main(args: Array[String]) {

val fruit = Set("apples", "oranges", "pears")

val nums: Set[Int] = Set()

println( "Head of fruit : " + fruit.head )

println( "Tail of fruit : " + fruit.tail )

println( "Check if fruit is empty : " + fruit.isEmpty )

println( "Check if nums is empty : " + nums.isEmpty )

}

}

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

### Command

\>scalac Demo.scala\>scala Demo

### Output

Head of fruit : apples

Tail of fruit : Set(oranges, pears)

Check if fruit is empty : false

Check if nums is empty : true

## Concatenating Sets

You can use either **++** operator or **Set.++()** method to concatenate two or more sets, but while adding sets it will remove duplicate elements.

The Following is the example to concatenate two sets.

### Example

object Demo {

def main(args: Array[String]) {

val fruit1 = Set("apples", "oranges", "pears")

val fruit2 = Set("mangoes", "banana")

// use two or more sets with ++ as operator

var fruit = fruit1 ++ fruit2

println( "fruit1 ++ fruit2 : " + fruit )

// use two sets with ++ as method

fruit = fruit1.++(fruit2)

println( "fruit1.++(fruit2) : " + fruit )

}

}

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

### Command

\>scalac Demo.scala\>scala Demo

### Output

fruit1 ++ fruit2 :

Set(banana, apples, mangoes, pears, oranges)

fruit1.++(fruit2) : Set(banana, apples, mangoes, pears, oranges)