Kotlin Function

Function is a group of inter related block of code which performs a specific task.

Function is used to break a program into different sub module.

It makes reusability of code and makes program more manageable.

In Kotlin, functions are declared using **fun** keyword.

There are two types of functions depending on whether it is available in standard library or defined by user.

- Standard library function
- User defined function

Standard Library Function

Kotlin Standard library function is built-in library functions which are implicitly present in library and available for use.

For example

```
fun main(args: Array<String>){
     var number=25
     var result=Math.sqrt(number.toDouble())
     print("Square root of $number is $result")
}
```

- Here, sqrt() is a library function which returns square root of a number (Double value).
- print() library function which prints a message to standard output stream.

User defined Function

User defined function is a function which is created by user. User defined function takes the parameter(s), perform an action and return the result of that action as a value.

Kotlin functions are declared using the fun keyword.

For example:

```
fun functionName(){
// body of function
}
```

Kotlin simple function example

```
fun main(args: Array<String>){
    sum()
    print("codeaftersum")
}

fun sum(){
    var num1=5
    var num2=6
    println("sum="+(num1+num2))
}
```

Kotlin Parameterize Function and Return Value

Functions are also takes parameter as arguments and return value. Kotlin functions are defined using Pascal notation, i.e. name:type (name of parameter and its type). Parameters in function are separated using commas.

If a function does not returns any value than its return type is Unit. It is optional to specify the return type of function definition which does not returns any value.

```
fun functionName(number1:Int, number2:Int){
......
}
......
functionName(value1,value2)
......
```

Kotlin parameterize function example

```
fun main(args: Array<String>){
    val result=sum(5,6)
    print(result)
}
fun sum(number1:Int,number2:Int):Int{
    val add=number1+number2
    return add
}
```

Kotlin Recursion Function

Recursion function is a function which calls itself continuously.
This technique is called recursion.

Syntax

```
fun functionName(){
      ......
      functionName()//callingsamefunction
}
```

Kotlin recursion function example 1: Finite times

Let's see an example of recursion function printing count.

```
varcount=0
fun rec(){
      count++;
      if(count<=5){
            println("hello"+count);
            rec();
      }
}

fun main(args: Array<String>){
      rec();
}
```

Kotlin recursion function example 2: Factorial Number

```
fun main(args: Array<String>){
      val number=5
      val result:Long
      result=factorial(number)
      println("Factorialof$number=$result")
}

fun factorial(n: Int): Long{
      return if(n==1){
            n.toLong()
      }
      else{
            n*factorial(n-1)
      }
}
```

Working process of above factorial example

```
factorial(5)
     factorial(4)
          factorial(3)
               factorial(2)
                    factorial(1)
                         return1
               return2*1=2
          return3*2=6
     return4*6=24
return5*24=120
```

Kotlin Tail Recursion

General Recursion

Let's see an example of calculating sum of nth (100000 larger number) using general (normal) recursion.

```
     fun main(args: Array<String>){
          var result=recursiveSum(100000)
          println(result)
     }
     fun recursiveSum(n: Long): Long{
          return if(n<=1){
               n
          }else{
               n+recursiveSum(n-1)
          }
     }
```

Tail Recursion

Tail recursion is a recursion which performs the calculation first, then makes the recursive call. The result of current step is passed into the next recursive call.

Tail recursion follows one rule for implementation.
This rule is as follow:
The recursive call must be the last call of the method.
To declare a recursion as tail recursion we need to use tailrec modifier before the recursive function.

Kotlin Tail Recursion Example 1: calculating sun of nth(100000) number

```
fun main(args: Array<String>){
     var number=100000.toLong()
     var result=recursiveSum(number)
     println("sum of up to $number number = $result")
}

tailrec fun recursiveSum(n:Long,semiresult:Long=0):Long{
     return if(n<=0){
          semiresult
     }else{
          recursiveSum((n-1),n+semiresult)
     }
}
```

Kotlin Tail Recursion Example 2:

calculating factorial of number

```
fun main(args: Array<String>){
     val number=4
     val result:Long
     result=factorial(number)
          println("Factorialof$number=$result")
}

tailrec fun factorial(n:Int, run:Int=1):Long{
     return if(n==1){
          run.toLong()
     }else{
          factorial(n-1,run*n)
     }
}
```

Kotlin Default Argument

Kotlin provides a facility to assign default argument (parameter) in a function definition.

If a function is called without passing any argument than default argument are used as parameter of function definition. And when a function is called using argument, than the passing argument is used as parameter in function definition.

Default argument example 1:
passing no argument in function call

```
fun main(args: Array<String>){
     run()
}
fun run(num:Int=5,latter:Char='x'){
     print("parameter in function definition $num and
$latter")
}
```

Default argument example 2:
passing some argument in function call

```
fun main(args: Array<String>){
     run(3)
}
fun run(num:Int=5,latter:Char='x'){
     print("parameter infunction definition $num and
$latter")
}
```

Default argument example 3:
passing all argument in function call

```
funmain(args:Array<String>){
     run(3,'a')
}
fun run(num:Int=5,latter:Char='x'){
     print("parameter in function definition $num and
$latter")
}
```

Kotlin Named Argument

Before we will discuss about the named parameter, let's do some modify in the above program.

For example:

```
fun main(args: Array<String>){
     run('a')
}
fun run(num:Int=5,latter:Char='x'){
     print("parameter in function definition $num and
$latter")
}
```

Output:

Error: Kotlin: The character literal does not conform to the expected type Int

Kotlin Named Argument Example

```
fun main(args: Array<String>){
run(latter='a')
}
fun run(num:Int=5,latter:Char='x'){
     print("parameter in function definition $num and
$latter")
}
```

Unit-returning functions

- If a function does not return any useful value, its return type is Unit.
- Unit is a type with only one value - Unit.
- This value does not have to be returned explicitly:

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
    else
        println("Hi there!")
    // `return Unit` or `return` is optional
}
```

The Unit return type declaration is also optional. The above code is equivalent to:

```
fun printHello(name: String?) { ... }
```

Single-Expression functions

When a function returns a single expression, the curly braces can be omitted and the body is specified after a = symbol:

```
fun double(x: Int): Int = x * 2
```

Explicitly declaring the return type is optional when this can be inferred by the compiler:

```
fun double(x: Int) = x * 2
```

Kotlin Lambda Function

- Lambda is a function which has no name.
- Lambda is defined with a curly braces {} which takes variable as a parameter (if any) and body of function.
- The body of function is written after variable (if any) followed by -> operator.

Syntax of lambda

```
{ variable -> body_of_function}
```

Normal function: addition of two numbers

In this example, we create a function addNumber() passing two arguments (a,b) calling from the main function.

```
fun main(args: Array<String>){
    addNumber(5,10)
}
fun addNumber(a: Int, b: Int){
    val add = a + b
    println(add)
}
```

Lambda function: addition of two numbers

The above program will be rewritten using lambda function as follow:

```
fun  main(args: Array<String>){
     val  myLambda:(Int)->Unit={
               s:Int->println(s)
          }//lambdafunction
          addNumber(5,10,myLambda)
}
fun addNumber(a:Int,b:Int, myLambda:(Int)->Unit){
     //highlevelfunctionlambdaasparameter
     val add=a+b
     myLambda(add)//println(add)
}
```

Higher order function

High order function (Higher level function) is a function which accepts function as a parameter or returns a function or can do both.

Means, instead of passing Int, String, or other types as a parameter in a function we can pass a function as a parameter in other function.

```
fun myFun(
          org:String,
          portal:String,
          fn:(String,String)->String):Unit{
               val result=fn(org, portal)
               println(result)
          }
```

In this above example, we defined a function myFun() with three parameters.
The first and second parameter take String and the third parameter as a type of function from String to String.
The parameter String to String type means function takes string as an input and returns output as string types.

To call this above function, we can pass function literal or lambda.
For example:

```
fun myFun(org:String,
          portal:String,
          fn:(String,String)->String):Unit{
               valresult=fn(org,portal)
               println(result)
          }

fun main(args: Array<String>){
     val fn:(String,String)->String =
                         {org,portal
                              ->"$orgdevelop$portal"}
     myFun("sssit.org","javatpoint.com",fn)
}
```

Inline Function

- An inline function is declare with a keyword inline.
- The use of inline function enhances the performance of higher order function.
- The inline function tells the compiler to copy parameters and functions to the call site.
- The virtual function or local function cannot be declared as inline.

Following are some expressions and declarations which are not supported anywhere inside the inline functions:

- Declaration of local classes
- Declaration of inner nested classes
- Function expressions
- Declarations of local function
- Default value for optional parameters

Let's see the basic example of inline function:

```
fun main(args: Array<String>){
     inlineFunction(
          {println("calling inline functions")}
     )
}

inline fun inlineFunction(myFun:()->Unit){
     myFun()
     print("code inside inline function")
}
```

Non local control flow

From inline function, we can return from lambda expression itself.

This will also lead to exit from the function in which inline function was called.

The function literal is allowed to have non local return statements in such case.

```
fun main(args: Array<String>){

inlineFunction({println("calling inline functions")
          return},
          {println("next parameter in inline functions")}
     )
}

inline fun inlineFunction(
          myFun:()->Unit,
          nxtFun:()->Unit){
               myFun()
               nxtFun()
               print("code inside inline function")
          }
```

crossinline annotation

To prevent *return* from lambda expression and inline function itself, we can mark the lambda expression as crossinline.

This will throw a compiler error if it found a return statement inside that lambda expression.

```
fun main(args: Array<String>){

     inlineFunction({println("calling inline functions")
          return //compiletimeerror
     },{println("nextparameterininlinefunctions")})
}
inline fun inlineFunction(crossline myFun:()->Unit,
                    nxtFun:()->Unit){
     myFun()
     nxtFun()
     print("codeinsideinlinefunction")
}
```

noinline modifier

In inline function, when we want some of lambdas passed in inline function to be an inlined, mark other function parameter with noinline modifier.

This is used to set expressions not to be inlined in the call.

```
Fun main(args: Array<String>){
     inlineFunctionExample(
          {println("calling inline functions")},
          {println("next parameter in inline functions")}
     )
     println("this is main function closing")
}

inline fun inlineFunctionExample(
          myFun:()->Unit,
          noinline nxtFun:()->Unit){
               myFun()
               nxtFun()
               println("code inside inline function")
          }
```

Output:

```
calling inline functions
next parameter in inline functions
code inside inline function
this is main function closing
```

https://www.javatpoint.com/kotlin-default-and-named-argument