Kotlin Tips and Tricks for Efficient Programming

Kotlin -a modern statically typed programming language. It is expressive and concise. Kotlin offers many features that make code lesser and readable. In this blog, we will re-write the code in **the Kotlin way** and see how Kotlin makes code more concise and readable.

Scope functions: The purpose of scope functions are to provide a block with the context of a given object. There are five of them: `let`, `run`, `with`, `apply`, and `also`.

let: It can be used to invoke one or more functions on results of call chains.

Without let

```
val basicSalary = getBasicSalary()
calculateHRA(basicSalary)
calculateDA(basicSalary)
calculateTA(basicSalary)
```

With let

```
getBasicSalary().let{
    calculateHRA(it)
    calculateDA(it)
    calculateTA(it)
}
```

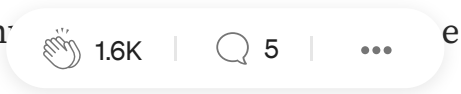
apply: Use it when you accessing multiple properties of an object.

Without apply

```
val user = User()
user.name = "Simform"
user.email = "simformsolutions@gmail.com"
```

With apply

```
val user = User().apply{
    name = "Simform"
    email = "simformsolutions@gmail.com"
}
```

with: It is also used to call m 

e object.

Without

```
calculateHRA(basicSalary)
calculatePF(basicSalary)
```

Using with

```
with(basicSalary){
    calculateHRA(this)
    calculatePF(this)
}
```

run: It is also a scope function and useful when you want to do object initialization and some computation of return value.

```
val user = User().run{
    name = "Simform"
```

```
    formatAddress()
}
```

also: Use it for actions that need a reference rather to the object than to its properties and functions, or when you don't want to shadow `this` reference from an outer scope.

```
val numbers = mutableListOf("one", "two", "three")
numbers.also {
    println("The list elements before adding new one:$it")
}
.add("four")
```

Data class(POJOs/POCOs):

```
data class User(val name:String, val email:String)
```

Data class provides the following functionalities by default.

1. getters (and setters in case of `var` s) for all properties
2. `equals()`
3. `hashCode()`
4. `toString()`
5. `copy()`
6. `component1()` , `component2()` , …, for all properties (see Data classes)

Default parameters: Default parameters allow us to give default values to a function's parameter. So when calling this function if you do not pass the second parameter then it takes the default value of 0 and if you pass any value then the function will use that passed value. This will allow us to use method overloading without having to write multiple method signatures for different numbers of arguments.

```
fun calculateSalary(var sal:Int, var perentHRA: Int = 8) : Double{
//steps for performing HRA Calculation
}
```

```
fun add(var one: Int, var two: Int = 0)
```

```
add(2,3)# add(2)
calculateHRA(5000, 6)
calculateHRA(4000)
```

Extension functions: Extension functions are functions that are defined for specific data types and calling these functions is the same as member functions with the (.) operator. Extension function is used to extend the functionality of a specific class by creating functions that are only called upon that class object and are user-defined. With the extension functions, we can add more functions to the classes that we don't have access to like built-in classes (Ex. Int, String, ArrayList, etc..). Kotlin has many inbuilt extension functions like *toString(), filter(), map(), toInt()* etc..

```
fun Int.returnSquare(): Int { return this * this }

//Calling the extension function
val number = 5
println("Square is ${number.returnSquare()}")   // output: 25
```

Class StringUTil{

fun toUpperCase(){}
fun toLowerCase(){}
}

class PaySlip{

StringUtil.capitalizingNames(){
//write your logic for Capitalising
// names

emp?.fName, emp?.LastName
}
}

val emp = fetchEmployeeId(12)
val empFullName = StringUtil.
capitalizingNames(emp)

> ***Note:*** `this` *keyword refers to receiver object (In above case Int)*

null safe operator (?.): It is used on nullable type variables to safely use them in code and avoid `nullPoiterException`.

```
val files = getFiles()
print(files?.size)  //It prints size if files is not null
```

use **`let`** with **`?.`** to call multiple methods or execute statements if variable or expression is not null.

```
val user = getUser()
user?.let{
    //Execute when user is not null
    saveUserToDB(it)
}
```

Elvis operator(?:)

if(a>b)?return a : return b

Without

```
if(user.name != null) {
    userName = user.name
} else {
    throw IllegalStateException()
}
```

With

```
val userName = user.name?: throw IllegalStateException()
```

also, we can execute multiple statements if the value is null with *run*

```
user.name?.let{
   // Excecute if not null
} : run {
   //Execute if null
}
```

Single-expression functions: If a function just returns a value then you can write it in single-expression.

```
fun isOdd(number:Int) = if(number%2 ==0) false else true
```

when: It is similar to `switch` of java but it is more flexible.

```
when(number:Int) {
    5 -> "Greater than five"
    in 6..10 -> "In range of 6 to 10"
    else -> "This is else"
}
```

Range operator(..)

```
for( i in 1..10) // 1 to 10
for( i in 1 until 10) // 1 to 9 (Does not include 10)
for( i in 1..10 step 2) // 1,3,5,7,9
for( i in 10 downTo 1) // 10,9,8....,1
```

for(i=10;i<=1;i--){
//starts 10 upto 1
}

Check Instance with *is* operator

is and !is
val emp = Employee()
emp.name
if(emp is 10 is a Int?
!is -> negation

if(emp is Employee){
}

if(dept !is Employee){
}

```
if(10 is Int)  // true
if(10 is Boolean) // false
if("string" !is Int) // true
```

Lambda functions: Lambda functions are anonymous functions that we can treat as values. We can pass lambda functions as arguments and store them as variables.

```
{argumentName: argumentType -> // lamda body}
```

Argument name and type are optional, we can omit that. Lambda body is required. The type of the last line of lambda body is the return type of lambda.

```
val double: (Int)-> Int = {number:Int -> number * number}
println(double(5)) // output: 25
```

Here this lambda takes one integer as an argument and returns multiplication with the same number as Integer.

Some cool kotlin built-in functions:

filter : Filters the list, set, or map with the given predicate and returns a list, set, or map with elements that match the predicate. Find all filter variants here.

```
val numbers = listOf(1,2,3,4,5)

numbers.filter {element-> element%2 == 0 } // output list [2,4]

numbers.filterIndexed{index,element->(index != 0) && (element< 5)}
// [2,3,4]

numbers.filterNot {element-> element <= 3 } // [4,5]
```

map : Applies given predicate or transformation function to each element of the collection and returns a new collection.

```
val numbers = listOf(1,2,3,4,5)

numbers.map { it * 3 } // output list [3,6,9,12,15]

numbers.mapIndexed { index,value -> value * index } // [0,2,6,12,20]

//following functions are used to get non-null values

numbers.mapNotNull {value-> if ( value == 2) null else value * 3 }
// [3,9,12,15]

numbers.mapIndexedNotNull {
    index, value -> if (index == 0) null else value * index
}
// [2,6,12,20]

val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3,
"key11" to 11)

numbersMap.mapKeys { it.key.uppercase() }
//{KEY1=1,KEY2=2,KEY3=3,KEY11=11}

println(numbersMap.mapValues { it.value + it.key.length })
//{key1=5, key2=6, key3=7, key11=16}
```

zip : It creates a list of pairs with elements of the same index from the given two lists.

```
val colors = listOf("red", "brown", "grey")
val animals = listOf("fox", "bear", "wolf")
println(colors.zip(animals))
```

```
//Output:   [(red,fox),(brown,bear),(grey,wolf)]
```

You can create two lists from the list of pairs by applying ***unzip*** .

```
val numberPairs = listOf("one" to 1, "two" to 2, "three" to 3,
"four" to 4)
println(numberPairs.unzip())

//Output: ([one, two, three, four], [1, 2, 3, 4])
```

var helloStr: String = Hello Mr John Welcome to Kotlin
helloStr.joinToString('#')
println(helloStr)

joinToString() : This will create a string with all elements appended.

joinTo() : This will create a string with all elements appended and append that to the given string in the argument.

```
val numbers = listOf("one", "two", "three", "four")
//Output: one, two, three, four

val listString = StringBuffer("The list of numbers: ")
numbers.joinTo(listString)
//Output: The list of numbers: one, two, three, four
```

flatten() : It creates one list from the list of lists.

```
val listOfList = [[1,2],[3,4,5],[6,7]]
println(listOfList.flatten())   //Output: [1,2,3,4,5,6,7]
```

any : It takes lambda and checks whether the given predicate in lambda matches any of the elements from the list. If yes then it returns true otherwise false.

all : If the given predicate matches all the elements of a collection then returns true otherwise false.

none : If the given predicate does not match any of the elements from the collection then it returns true otherwise false.

```
val numbers = listOf("one", "two", "three", "four")
```

```
numbers.any { it.endsWith("e") } // Output: true
numbers.none { it.endsWith("a") }  // true
numbers.all { it.endsWith("e") } // false
```

partition : It will return pair of lists one with elements that match the condition and one with elements that do not match the condition.

slice : It will create a list with the given index.

chunked : It will also create a list of lists but with the given size.

```
val numbers = listOf("one", "two", "three", "four")

numbers.partition { it.length > 3 }
// (["one","two"],["three","four"])

numbers.slice(1..3)  // ["two","three","four"]

numbers.chuncked(2) // [["one","two"],["three", "four"]]
```

take : It will get the specified number of elements starting from the first.

takeLast : It will get the specified number of elements starting from the last.

drop : It will take all the elements except a given number of first elements.

dropLast : It will take all the elements except a given number of last elements. For more methods see here.

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.take(3))  // [one, two, three]
println(numbers.takeLast(3)) //[four, five, six]
println(numbers.drop(1))  // [two, three, four, five, six]
println(numbers.dropLast(5)) // [one]
```

groupBy : It takes a lambda function and returns a map. In a result map, keys will be the result of lambda functions and values will be the corresponding list element on which the lambda function is applied. It is used to group list elements with specific conditions.

```
val numbers = listOf("one", "two", "three", "four", "five")

println(numbers.groupBy { it.first().uppercase() })
```

```
//Output: {O=[one], T=[two, three], F=[four, five]}
```

average : This will return the average of the elements of the list.
sum : This will return the sum of all the elements of the list.
count : This will return the count of the elements in the list.
minOrNull : This will return the smallest value from the list or return null when the list is empty.
maxOrNull : This will return the largest value from the list or return null when the list is empty.

```
val numbers = listOf(6, 42, 10, 4)

println("Count: ${numbers.count()}")    // Output : 4
println("Max: ${numbers.maxOrNull()}")  //42
println("Min: ${numbers.minOrNull()}")  //4
println("Average: ${numbers.average()}")//15.5
println("Sum: ${numbers.sum()}")        //62
```

There are some collection-specific functions also you can find them:
1. list specific functions
2. set specific functions
3. map specific functions

Avoid indexOutOfBound error with these functions:

elementAtOrNull() : Returns null when the specified position is out of the collection bounds.
elementAtOrElse() : Returns the result of the lambda on the given value when the specified position is out of the collection bounds.

Avoid nuberFromatException with these functions:

toIntOrNull() : Converts string to Int and returns null if an exception occurs.
toDoubleOrNull() : Converts string to double and returns null if an exception occurs.
toFloatOrNull() : Converts string to float and returns null if an exception occurs.

> ***Note:*** *You need to handle null values by yourself otherwise nullPointerException will be thrown.*

So these are some cool features of Kotlin and some are still not included in this blog. You can find them here. If this blog helped you to make your kotlin code more concise and readable then please don't forget to give claps 👏 and share this with your fellow coder friends. Let's write the code in Kotlin way :)

Kotlin Kotlin Beginners Extension Functions Android Kotlin Lambdas