Fresher Android

Kotlin Basics – Day 2


FPT
GLOBAL SMART
TECHNOLOGIES
Functions

1. Default arguments

- Function parameters can have default values. Default values are defined using the **=** after type along with the value:

```
fun lockTheDoor(timeout: Int = 3000) {
        /*...*/
}
```

- When calling the function, the parameters that have default value are not needed to set value.
- The default arguments can not be overridden by the child class:

```
open class A {
open fun foo(i: Int = 10) { /*...*/ }
}

class B : A() {
override fun foo(i: Int) { /*...*/ }  // no default value allowed
}
```

2. Named arguments

- Function parameters can be named when calling functions, then the code much more readable:

```
fun reformat(str: String,
        normalizeCase: Boolean = true,
        upperCaseFirstLetter: Boolean = true,
        divideByCamelHumps: Boolean = false,
        wordSeparator: Char = ' ') {
        /*...*/
        }
```


- When calling:
  -

```
reformat(str,
        normalizeCase = true,
        upperCaseFirstLetter = true,
        divideByCamelHumps = false,
        wordSeparator = '_‘
        )
```

3. Unit-returning

- If a function does not return any value, its return type is Unit:

```
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.

4. Single-expression

- 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 {
    return x * 2
}
```

- Could be:

```
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
```

5. Variable number of arguments

- A parameter of a function (normally the last one) may be marked with vararg modifier:

```
fun asListOf(vararg strings: String): ArrayList<String> {
    val result = ArrayList<String>
    for (string in strings) {
        result.add(string)
    }
    return result
}
```

- When calling:

```
val listString = asListOf(“aaaa”, “bbbb”, “cccc”)
```

6. Extension functions

- Extension function ability to extend a class with new functionality without having to inherit the origin class.
- Example: we have a function that can plus 2 numbers

```
fun plus(number1: Int, number2: Int): Int {
        return number1 + number2
}

val number = plus(1, 2)
```

- How about to make the class **Int** to have function **plus:**

```
fun Int.plus(number: Int): Int {
        return this + number
}
```

- When calling:

```
val numberA = 2
println(numberA.plus(3)) // result is 5
println(5.plus(2)) // result is 7
```

7. Infix notation

- Infix notation is a method to simple the calling of a funtion
- Example: In previous slide, we have function plus:

```
fun Int.plus(number: Int): Int {
    return this + number
}
```

- We can using keyword **infix** to mark the function as infix function

```
infix fun Int.plus(number: Int): Int {
    return this + number
}
```

- When calling, we no need to use the dot

```
val result = 2 plus 3  // result = 5
```

- The infix function must be a member functions or extension functions.
- The infix function must have a single param and no default value.

8. Local functions

- A function inside another function is called a local function:

```
fun getStudentName(): String {
        fun normalize(str: String): String {
                return “Student $str”
        }
        return normalize(name) + “ (PTG)”
}
```

- The scope of the local function is inside the parent function.

9. Generic functions

- Functions can have generic parameters which are specified using angle brackets before the function name.
- Example:

```
fun asListOf(vararg strings: String): ArrayList<String> {
        val result = ArrayList<String>
        for (string in strings) {
                result.add(string)
        }
        return result
}
```

- How about if using below with another types?

```
fun <T> asListOf(vararg params: T): ArrayList<T> {
        val result = ArrayList<T>
        for (item in params) {
                result.add(item)
        }
        return result
}
```

- Calling:

```
val list = asListOf(1, 2, 3)
val list2 = asListOf("a", "bb", "ccc")
```

Functions

1. Default arguments
2. Named arguments
3. Unit-returning
4. Single-expression
5. Variable number of arguments
6. Extension functions
7. Infix notation
8. Local functions
9. Generic functions

GLOBAL SMART TECHNOLOGIES | FPT

Lambdas

1. Function types

- Function can be declare as a variable and it can have a type:
  - ✓ **()->Unit**: declare a function that have no parameters and returned value
  - ✓ **(Int)->Int**: declare a function that have a integer param, and returned result is a integer
  - ✓ **()->()->Unit**: declare a function that have no param, and returned another function with type is ()->Unit

- A function type can be used as a interface:

```
class MyFunction: ()->Unit {
                    override fun invoke() { println("I am called") }
}
val function = MyFunction()
function()
```

- A function type can be used as a variable, property or arguments:

```
val greet: ()->Unit
val square: (Int)->Int
val producePrinter: ()->()->Unit
```

2. Higher-Order Functions

- A higher-order function is a function that takes functions as parameters, or returns a function.

```
fun doSomethingWithNumber(number: Int, receiver: (String?) -> Unit) {
    val num = number + 10
    receiver(num.toString())
}
```

- Calling:

```
doSomethingWithNumber(2) {
        println(it)  // Function Anonymous
}

doSomethingWithNumber(2, {
        println(it) // Function Anonymous
 })

doSomethingWithNumber(2, ::println)  // ::println is function reference of println()
```

3. Lambda Expressions and Anonymous Functions

- Lambda expression is a way to simple a function declaration:

```
val helloFunc: ()->Unit = {
                println("Hello")
}

val squareFunc: (Int)->Int = {x ->
                x*x
}
```

- Anonymous function is another way to define a function:

```
val helloFunc = fun() {
                println("Hello")
}

val squareFunc = fun(x: Int) = x*x
```

Lambdas

1. Function types
2. Higher-Order Functions
3. Lambda Expressions and Anonymous Functions

GLOBAL SMART TECHNOLOGIES

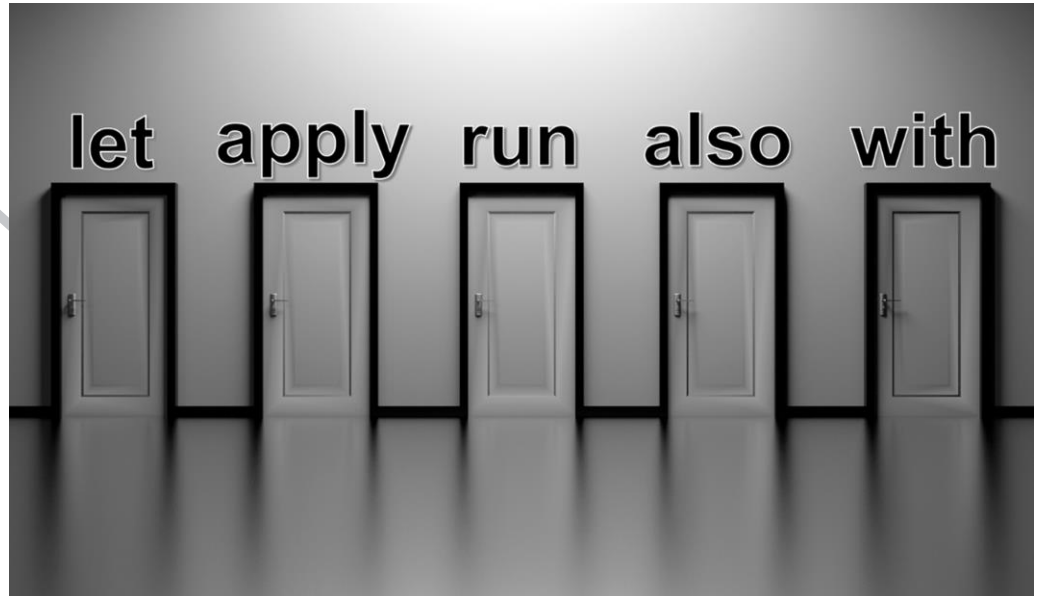
Scope Functions

1. Scope functions

- Kotlin defines several functions whose sole purpose is to execute a block of code within the context of an object, they are scope functions.
- When calling these functions with a object, it will create a temporary block, in this block we can access the object without its name.
- We have five scope functions: *let, run, with, apply, and also.*

2. Compare: let, apply, run, also, with

- To compare the scope functions: let, apply, run, also, with; we will base on the following criteria:
  - ✓ extension function
  - ✓ it and this
  - ✓ return



3. Extension function vs Normal function

```
val str = "Test string"
str.run {
        this.trim()
        val last = this.last()
}
with(str) {
        this.trim()
        val last = this.last()
}
run {
        str.trim()
        val last = this.last()
}
```

```
val str: String? = "Test string"
str?.run {
        this.trim()
        val last = this.last()
}
with(str) {
        this?.trim()
        val last = this?.last()
}
run {
        str?.trim()
        val last = this?.last()
}
```


- ➔ **with, run are normal function**
- ➔ **let, apply, run, also are extension function**

4. Using “it” vs “this”

```
val str = “Test string”

str.let {
            it.trim()
    println(it)
}

str.apply {
            trim()
    println(this)
}
```


- ➔ **it is current object, block current is class of object: let, also**
- ➔ **do not have it, block current is class of object: apply, run, with**

#5. Return this or return anything

```
val student = Student()

student.apply {
                println(name)
}.birth // get birth OK

student.apply {
                println(name)
                “Test String”
}.trim() // Error: returned is student, not string
```

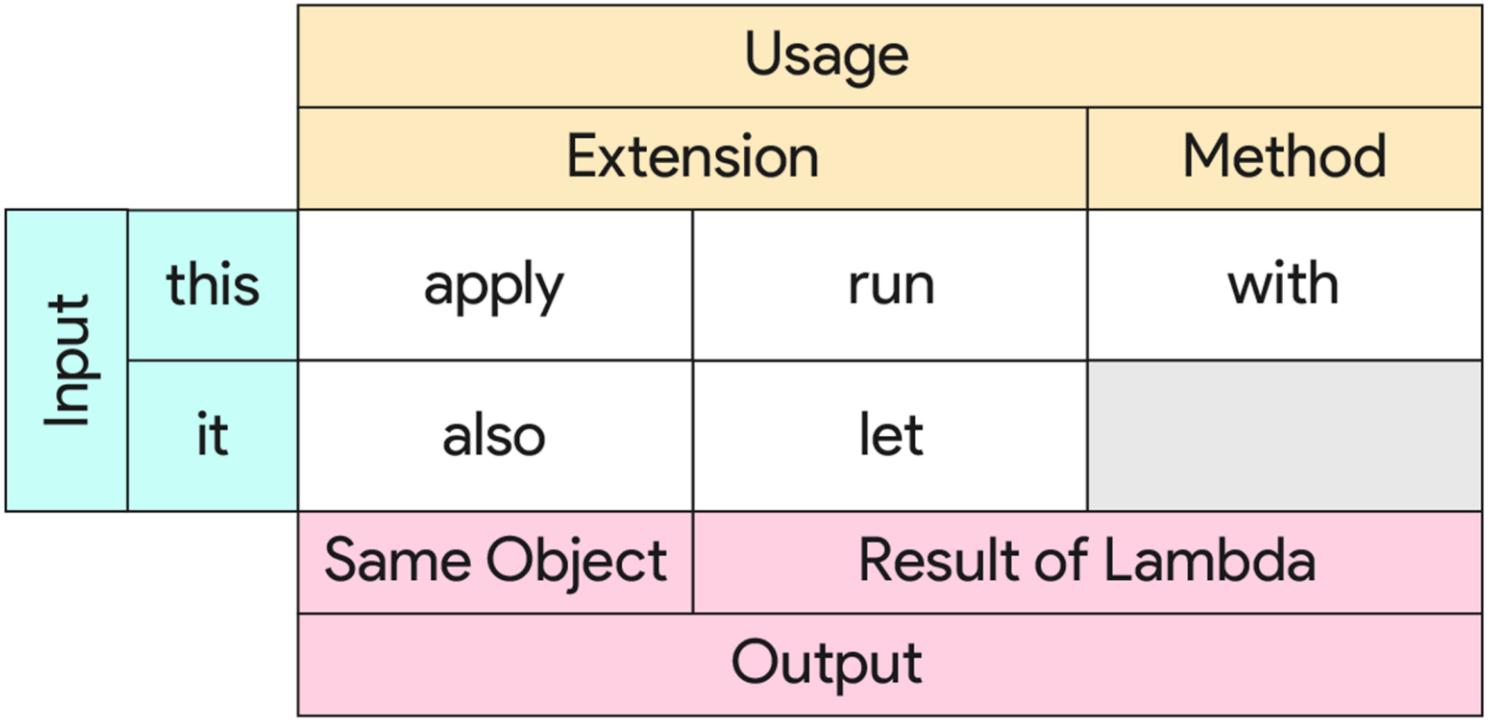
➔ **return current object: apply, also**
➔ **return anything: let, run, with**

```
val student = Student()

student.run {
                println(name)
}.birth // Error: returned nothing

student.run {
                println(name)
                “Test String”
}.trim() // trim “test string” OK

student.run {
                println(name)
                this
}.birth // get birth OK
```

Scope Functions

1. Scope functions
2. Compare: let, apply, run, also, with
3. Extension function vs Normal function
4. Using “it” vs “this”
5. Return this or return anything
6. Compare: let, apply, run, also, with

6. Compare: let, apply, run, also, with



Lesson Summary

1. Functions
2. Lambdas
3. Scope Functions

Thank you