Android Development with Kotlin

Higher Order Functions in Kotlin

Coding Ninjas

KOTLIN HIGHER ORDER FUNCTIONS:

Kotlin language has a fantastic support for functional programming. Kotlin functions can be stored in variables and data structures, passed as arguments to and returned from other higher-order functions. Kotlin functions are first-class, which means that they can be stored in variables and data structures, passed as arguments to and returned from other higher-order functions. You can operate with functions in any way that is possible for other non-function values.

Lambda Expressions:

Lambdas Expressions are essentially anonymous functions that we can treat as values – we can, for example, pass them as arguments to functions, return them, or do any other thing we could do with a normal object.

```
val square : (Int) -> Int = { value -> value * value }
val nine = square(3)
```

- **In (Int) -> Int**
- **(Int)** represents input int as a parameter.
- **Int** represents return type as an int.

Higher-Order Function:

In Kotlin, a function which can accept a function as parameter or can return a function is called Higher-Order function. Instead of Integer, String or Array as a parameter to function, we will pass anonymous function or lambdas. Frequently, lambdas are passed as parameter in Kotlin functions for the convenience. **It's a function which can take do two things:**

- Can take functions as parameters
- Can return a function

```
fun passMeFunction(abc: () -> Unit) {// example for taking function
as parameter
  // I can take function
  // do something here
  // execute the function
  abc()
}
```

- **This takes a function abc: () -> Unit**
- **abc is just the name for the parameter.** It can be anything. We just need to use this when we execute the function.
- **() -> Unit**, this is important.
- () represents that the function takes no parameters.
- **Unit** represents that the function does not return anything.

Coding Ninjas

- So, the **passMeFunction** can take a function which takes zero parameters and do not return anything.

```
fun add (a: Int, b: Int): Int {
  return a + b
}

fun returnMeAddFunction(): ((Int, Int) -> Int) {
   // can do something and return function as well
   // returning function
   return ::add
      }
```

- ((Int, Int) -> Int)
- **(Int, Int)** means that the function should take two parameters both as the int.
- **Int** means that the function should return value as an int.

Now, we can call the **returnMeAddFunction**, get the add function and call it like below:

```
val add = returnMeAddFunction()

val result = add(2, 2)
```

Anonymous functions:

Anonymous functions are function declarations without a name. The filter function, an anonymous function should look something like this:

```
val filter = fun(x : Int) : Boolean = x < 2
```

Kotlin Closures:

Closures are functions that can access and modify properties defined outside the scope of the function. The following closure function is a high order function that calculates the sum of all elements of the list and updates a property defined outside the closure.

```
var res = 0
myList = listOf(1,2,3,4,5,6,7,8,9,10)
myList.forEach { res+=it }
println(res) //prints 55
```

Inline Functions:

Basically, inline tells the compiler to copy these functions and parameters to call site. Similarly, **inline keyword** can be used with properties and property accessors that do not have backing field.

```
inline fun someMethod(a: Int, func: () -> Unit):Int {
func()
return 2*a
}
```

Coding Ninjas

Noinline Modifier:

Say you have multiple lambdas in your inlined function and you don't want all of them to be inlined, you can mark the lambdas you don't want to be inlined with the noinline keyword.

```
inline fun higherOrderFunction(aLambda: () -> Unit, noinline
dontInlineLambda: () -> Unit, aLambda2: () -> Unit) {

    doSomething()

    aLambda()
    dontInlineLambda()//won't be inlined.
    aLambda2()

    doAnotherThing()

}
```

Crossinline Modifier:

The crossinline marker is used to mark lambdas that mustn't allow non-local returns, especially when such lambda is passed to another execution context such as a higher order function that is not inlined, a local object or a nested function. In other words, you won't be able to do a return in such lambdas.

```
inline fun higherOrderFunction(crossinline aLambda: () -> Unit) {

    normalFunction {
        aLambda() //must mark aLambda as crossinline to use in this
context.
    }

}

fun normalFunction(aLambda: () -> Unit) {

    return

}

fun callingFunction() {

    higherOrderFunction {

        return  //Error. Can't return from here.
    }

}
```

Coding Ninjas