Android Development with Kotlin

Introduction to Kotlin

Objects

Kotlin unlike Java, does not allow the programmer to use the primitive data types directly. Every Data Type primitive or non-primitive is wrapped up in a class which is used to create objects of its type. The primitive data types however, work under the hood i.e. they are used in the background of these classes to make use of their robust and fast processing capabilities. The objects in Kotlin, do not require a type-specifier at the time of declaration if they are being initialized. The type is defined implicitly.

```
var string1: String = "ABC"
var string2 = "XYZ"
/*both of the statements produce the same results but, type specification is redundant in the case of string1, as it is also being initialized as a string and the Kotlin compiler can infer its data-type from the initialized value itself. */
```

Mutability

Mutability refers to the ability of a property (or object) to change its value over the course of the program. Much like Kotlin, Java too offers the concept of mutability using the "final" keyword but the final keyword isn't robust enough as the specification of an object's mutability isn't a compulsion in Java. Kotlin however, makes sure that the user specifies the mutability of the object at the type of its creation. Kotlin specifies a variable as "val" or "var" at the time of its declaration, where val is used for immutable objects whereas var is used for the mutable ones.

```
var string = "ABC"
val number: Int = 20
string = "XYZ"          //the statement runs without any error
number = 100            //prompts an error as "number" is immutable
```

Another major differences between Kotlin and Java in terms of variable declarations is that simpler data types cannot be assigned to more complex types and explicit casting is required to achieve the same.

```
val x: Int = 20
val y: Long = x.toLong()
```

So, the major differences in terms of mutability between kotlin and java would be:

- The variables can be mutable and immutable. In Kotlin immutable values are preferred whenever possible. The fact that most parts of our program are immutable provides lots of benefits, such as a more predictable behavior and thread safety.
- Type casting is done automatically.
- Simpler numerical types cannot be assigned to more complex types. Explicit casting is required.

In general, variables in Kotlin provide much more flexibility, safety (due to the convention of using val whenever possible) and cleaner, more concise code.

Type Check and Smart Cast

When working with mixed types, we often need to know the type of an object at runtime so that we can safely cast the object to our desired type and call methods or access properties on it. In Kotlin, one can check whether an object is of a certain type at runtime by using the 'is' operator. In Java, the object's instance is tracked using "instanceof" operator and needs to be explicitly cast as the class of which the object is an instance of. This is automatically in Kotlin as Kotlin itself casts the object when using with "is" operator. This is known as Smart Cast. Following is an example that demonstrates the usage of is operator.

```
if(obj is String && obj.length > 0) {
      println("Found a String of length greater than zero -
      ${obj.length}")   //object is automatically cast to type String
}
```

Kotlin's Type Cast Operator "as" can be used to manually cast a variable to a target type. If the variable can't be cast to the target type then the cast operator throws an exception. That's why we call the as operator "Unsafe". Kotlin also provides a Safe cast operator as? that returns null instead of throwing a ClassCastException if the casting is not possible.

```
var obj: String?
(obj as String).length        //returns the ClassCastException
(obj as? String).length       //supresses the exception
```

Null Safety

Kotlin is a null safe language and provides two separate data types for each primitive as well as user-defined class, a nullable data-type and a non-nullable data-type. The nullable data-type is specified with a "?" at the end of the specified type.

```
var string1: String = "ABC"
var string2: String? = "XYZ"
string1 = null
string2 = null
//the first statement prompts an error but the second statement is
allowed as string2 is of a nullable data type, whereas string1 is not.
```

Null Safety in Kotlin can be achieved with the help of three operators, the safe call operator (?), the Elvis operator (?:) and the not-null assertion operator (!!) operator. To avoid null conflicts in Java, we use if conditions, this is achieved in Kotlin using the "?" operator, which returns the value of the methods being called if the object is non-null, and returns "null" otherwise. The Elvis operator "?:" is used to replace the "null" in the safe call operator with a custom expression. The safe call is followed by the Elvis operator, which is then followed by a custom expression.

```
val b: String? = null
println(b?.length)            //prints "null" as b is null
val l = b?.length ?: -1       //the value of l is -1
```

Accessing a nullable object without the safe call operator, prompts a compilation error. This error can be avoided by replacing the safe call operator with the not-null assertion operator. This operator is only used when we are certain that the value of our object isn't null or else, it may lead to a null-pointer exception.

```
println(b.length)              //prompts a compilation error
println(b!!.length)            //suppresses the compilation error
//the above statement in this case however, will prompt a run-time
null-pointer exception as b is null.
```

Functions

In programming, function is a group of related statements that perform a specific task. Functions are used to break a large program into smaller and modular chunks. Dividing a complex program into smaller components makes our program more organized and manageable. Furthermore, it avoids repetition and makes code reusable. Depending on whether a function is defined by the user, or available in standard library, there are two types of functions, namely, Kotlin Standard Library Function, and User-defined functions.
To define a function in Kotlin, fun keyword is used. Then comes the name of the function (identifier) which is then followed by the functional arguments in the braces and a return type.

```
//a function that adds two numbers and returns the type int
fun addNumbers(n1: Double, n2: Double): Int {
      ... .. ...
}
```

Void v/s Unit Class

To understand the use of Void in Kotlin, let's first review what is a Void type in Java and how it is different from the Java primitive keyword void. The Void class, as part of the java.lang package, acts as a reference to objects that wrap the Java primitive type void. It can be considered analogous to other wrapper classes such as Integer — the wrapper for the primitive type int.
Now, Void is not amongst the other popular wrapper classes because there are not many uses cases where we need to return it instead of the primitive void. But, in applications such as generics, where we can not use primitives, we use the Void class instead.

```
fun returnTypeAsVoidSuccess(): Void? {
    println("Function can have Void as return type")
    return null         //has to return a value
}
fun unitReturnTypeForNonMeaningfulReturns(): Unit {
    println("No meaningful return")
    //a return statement isn't mandatory
}
```

Loops

Loop is used in programming to repeat a specific block of code. In this article, you will learn to create while and do...while loops in Kotlin programming. Loop is used in programming to repeat a specific block of code until certain condition is met (test expression is false). Kotlin offers 3 types of loops while, do while and for. The syntax of the while loop is as follows:

```
while (testExpression) {
      // codes inside body of while loop
}
```

The do...while loop is similar to while loop with one key difference. The body of do...while loop is executed once before the test expression is checked. Its syntax is as follows:

```
do {
      // codes inside body of do while loop
}while (testExpression);
```

The for loop in Kotlin iterates through anything that provides an iterator. In Kotlin, for loop is used to iterate through ranges, arrays, maps and so on (anything that provides an iterator). The syntax of for loop in Kotlin is:

```
for (item in collection) {
      // body of loop
}
```

Conditionals

Conditionals are an integral part of flow control in programming languages. Kotlin provides two conditions, “if” expressions and “when” expressions. The if expressions work similar to that in Java but the “when” expressions are an apt replacement and an improvement upon the “switch” statements present in Java.
“when” matches its argument against all branches sequentially until some branch condition is satisfied. “when” can be used either as an expression or as a statement. If it is used as an expression, the value of the satisfied branch becomes the value of the overall expression. If it is used as a statement, the values of individual branches are ignored. (Just like with if, each branch can be a block, and its value is the value of the last expression in the block.) The else branch is evaluated if none of the other branch conditions are satisfied. If when is used as an expression, the else branch is mandatory, unless the compiler can prove that all possible cases are covered with branch conditions. The example of a when statement is as follows:

```
when (x) {
      0, 1 -> print("x == 0 or x == 1")
      2 -> print(“x == 2”)
      else -> print("otherwise")
}
```

Array and List

An array is a collection of a similar data type with a fixed number of elements. Arrays in Kotlin are size-fixed but mutable i.e., any index of an array can be reassigned a new value with ease but the number of elements within an array cannot be altered. An element in an array can be accessed using its index number but the reverse is not possible. The definition of an array is given as

```
val x = arrayOf(1, 2, 3)
```

A list is a collection of elements irrespective of their data types. Kotlin offers two types of lists i.e. immutable list and mutable list. Both the size and values of an immutable list cannot be altered unlike an array where only the size is fixed. On the other hand, both the size and values of a mutable list can be altered. Unlike arrays, the elements of the list itself can also be used to

access their respective indices, and to alter the list that they belong to. The definition of mutable and immutable lists in Kotlin is given as

```
val x = listOf(1, 2, 3)
val y = mutableListOf(1, 2, 3)
x.add(4)          //prompts an error as x is immutable
y.add(4)
```

Arrays do offer a higher processing speed with less memory storage but lists are a more intuitive data-type and perform better than arrays in most of the cases. Lists are widely used and are often preferred over arrays in real-life implementations.

Sets and Maps

Sets are generic unordered collection of elements that does not support duplicate elements. Methods in this collection support only read-only access to the set; read/write access is supported through the mutable sets. Sets can be defined in Kotlin as follows:

```
val words2 = setOf("pen", "cup", "dog", "pen", "spectacles")
```

Maps is a collection that holds pairs of objects (keys and values) and supports efficiently retrieving the value corresponding to each key. Map keys are unique; the map holds only one value for each key. Methods in this collection support only read-only access to the map; read-write access is supported through the mutable maps. Maps can be defined in Kotlin as follows:

```
val map = mapOf(1 to "Geeks", 2 to "for" , 3 to "Geeks")
```