Android Mobile Application Development
Week 2: Kotlin

History

- Kotlin was developed by JetBrains in July 2011
- In February 2012 Kotlin was open sourced
- Kotlin 1.0 was released in February 2016
- Google announced Kotlin support at Google IO in 5/17
- Integrated into Android Studio 3.0
- In May 2019 Google announced that Kotlin was now its preferred language for Android development.
    - Kotlin-first https://developer.android.com/kotlin/first

Kotlin

- Kotlin is a modern language that is expressive and concise
    - Reduces boilerplate code that exists in Java
- Kotlin code is intended to be easier to understand and write and safe
- Kotlin is fully interoperable with Java
    - Compiled Kotlin code generates the same bytecode as that generated by the Java compiler enabling projects to be built using a combination of Java and Kotlin code
    - Kotlin can call Java-based code and Java can call Kotlin-based code
- Kotlin is object-oriented
- Kotlin is statically and strongly typed
- Kotlin is cross platform

Variables and Constants

- Variables and constants allow us to store values to be used in our programs
    - Use `val` for a variable whose value never changes (immutable).
        - Trying to change the value will result in the error "Val cannot be reassigned".
    - Use `var` for a variable whose value can change. (mutable)
- Only use variables when the value will change, otherwise use constants.
- You can't change a constant into a variable or a variable into a constant.
- To evaluate the value of a variable in a statement use `$variable`
    - To evaluate an expression `${expression}`

Types

In Kotlin everything is an object. Kotlin has some basic types because they're common but they are really still objects with functions and properties.

All the number types, Int, Double (64 bit floating point numbers), Float (32 bit), Long, Short, and Byte can be converted to any other number type with member functions such as `toInt(), toFloat(), etc`.

Kotlin also has data types

- Char for a single character (can't be treated directly as numbers)
- String
    - Strings are immutable
    - The String class provides many functions for manipulating strings

    - You can concatenate strings using the `+` operator or string templates
    - Elements of a string are characters that can be accessed by its index: `s[i]`.
- Boolean
    - True or false

Type Inference

- If you provide an initial value for a variable or constant Kotlin can infer the type
- Kotlin is a statically-typed language so the type is resolved at compile time and can never change.
- Without an initial value for a variable or constant you must provide the type

Example:
https://pl.kotl.in/jfo0YJHn4
You will notice that Kotlin does not require semi colons.

```
var message : String
message = "Hello class"
println(message)
var age : Int = 20
println(age)
var name = "Aileen"
println(name)
firstName="Gail"  (error, needs var or val)
val firstName = "Gail"
firstName = "me"  (error, can't change a constant)
println(firstName)
name = 20  (error, wrong type)
println("Hi " + name)
println("Hi $name")
```

Operators

- Kotlin supports the standard arithmetic and comparison operators
    - Shortcut operators are supported
        - Increment x by 1: x++
        - Decrement x by 1: x--
- Assignment is =
- Equality is = =
- String concatenation: +
- Augmented assignment operators +=, -=
    - a=a+2 or a+=2
- Logical operators
    - Logical NOT: !
    - Logical AND: &&
    - Logical OR: ||
- Comparison operators <, >, <=, >=
    - Kotlin translates these calls to `compareTo()`
- Range operator ..
    - a..b defines a closed range that includes a and b.

Conditionals

In Kotlin the test condition must be in parenthesis. The body must be in curly brackets if there's more than one instruction.

```
val temp = 8
val coldtemp = 10
if (temp < coldtemp){
        println("It's cold outside")
} else {
        println("It's nice out")
}
```

Change temp to 18 to test the else statement.

Kotlin also supports conditional expressions

- Each conditional branch returns the result of the expression on its final line

```
val messageString: String = if (temp < coldtemp) {
     "It's cold outside"
     } else {
          "It's nice out"
     }
println(messageString)
```

In Kotlin the `if` statement is actually an expression as it returns a result. This means you can use `if` in expressions. This replaces the need for the ternary operator (?) shorthand as we have in other languages such as Java. Use conditional expressions instead.
If we omit the curly braces for the blocks because they contain only single statements it even looks a bit like the ternary operator. We can also remove the String type declaration because it can use type inference.

```
val messageString = if (temp < coldtemp) "It's cold outside" else
"It's nice out"
```

When

Kotlin doesn't have a switch statement, the `when` statement is used instead and has a similar structure.

- A `when` statement compares a value against possible matching branches
- Each branch in a `when` expression is represented by a condition, an arrow (`->`), and a result
- The test condition can be a single value, multiple values separated by commas, or can be a range
- Branches do not automatically fall through so you don't need a break in each case
- The value for a branch must be the same data type as the variable in the test condition
- You can also supply an else option to execute if no other matches apply

```
when (age) {
     0 -> println("You're a baby")
     1, 2, 3, 4 -> println("You're a wee bitty one")
     in 5..21 -> println("Enjoy school")
     in 22..55 -> println("Welcome to the real world")
     else -> println("I don't know what you're doing")
```

```
}
```

The `when` statement can also be used as an expression where each branch becomes the returned value of the expression. In this case the `when` statement MUST be exhaustive.

- An else class MUST be included when a when statement is used as an expression

```
var msg = when (age) {
     0 -> "You're a baby"
     1, 2, 3, 4 -> "You're a wee bitty one"
     in 5..21 -> "Enjoy school"
     in 22..55 -> "Welcome to the real world"
     else -> "I don't know what you're doing"
}
println(msg)
```

Loops

Kotlin does not support the traditional C (and Java) style `for` loop. Instead it uses a `for-in` loop.

- Iterates through any sequence of items that provides an iterator
    - collection
    - range
- Requires parenthesis around test expression

```
for (count in 0..10)
{
      println(count)
}

for (i in 6 downTo 0 step 2) {
      println(i)
}
```

While loops are similar to other languages

- While loops evaluate its condition at the start of each pass through the loop
- do-while loops evaluate its condition at the end of each pass through the loop

Null Safety

null values are useful when representing “no value” but can cause a null reference exception error (`NullPointerException` in Java) when code encounters a null value where one was not expected.

Kotlin has the concept of nullable types that doesn’t exist in most other languages (Swift includes an optional type). The purpose of nullable types is to provide a safe and consistent approach to handling situations where a variable may have a null value assigned to it. The objective is to avoid the common problem of code crashing with the null pointer exception errors that occur when code encounters a null value where one was not expected.

Kotlin variables by default cannot be assigned a null value. This helps avoid the dreaded null reference exception error (`NullPointerException` in Java).
For a variable to be able to be assigned a null value you must define it to be nullable by adding “?” after its type.

`var city: String = null` (error, null cannot be a value of a non-null type String)
`var city: String? = null`

Not null assertion

- The not null assertion !! operator converts any value to a non-null type
- Throws an `NullPointerException` error if it's null
- Shouldn't be used if there's any chance it's null

`println(city!!)` (error, NullPointerException)

Explicit null check
You need to check to see if a variable or property is null before using it. There are a few ways to check to see if a variable is null before accessing it.

```
if (city != null){
      println(city.uppercase())
} else {
      println("City is null")
}

city = "Kigali"
```

Safe Calls
A more efficient way to check for null is to use the safe call operator.

- Using the safe call operator ? the property or method is called only if the variable is not null
- Equivalent to the if/else null check
- Can be used in chains
- If it evaluates to null on the left side of an assignment the assignment is skipped

```
println(city?.uppercase())
```

To perform an operation only for non-null values you can use the safe call operator together with the let() method.

- The let() method will only execute an action on a reference that is not null
    - Checks to see if the value is null
    - Runs an action only if the value is not null

```
city?.let{println(city)}
```

The let() method converts the nullable type to a non-null variable named `it` that can be used in the action.

```
city?.let{println(it)}
```

Elvis operator

- The Elvis operator ?: provides a shorthand for an if/else null check

- returns the value on the left side if it's not null
- returns the value on the right side if the expression on the left is null
    - the expression on the right side is only evaluated if the left side is null

```
println(city ?: "Elvis sighting")
```

Initialization

Non-null types have to be initialized when they're declared as they must have a value when they're first accessed. But there are times you want to declare a non-null variable but its value won't be known until later in the program. You can use the `lateinit` modifier to indicate that the variable will be initialized later. This allows you to declare a non-null variable before it is initialized, but you must ensure that the variable is initialized before the variable is accessed otherwise the code will fail.

```
var time : String
println(time)
      Error: Variable 'time' must be initialized

lateinit var time : String
println(time)
      Exception in thread "main"
kotlin.UninitializedPropertyAccessException: lateinit
property time has not been initialized

lateinit var time : String
time = "now"
println(time)
```

Functions

Rather than repeating the same series of expressions each time that you need to perform a task, you can wrap the expressions in a function and call that function instead.

In Kotlin functions can be written in three places

1. Outside a class (top-level functions)
    a. Scope is the package
    b. Useful for utility functions that don't belong in any one class
    c. Reduces the need to create a helper class just for utility functions
    d. Don't need a class or object to call these functions
2. Inside a class (member functions)
    a. Scope is the class
3. Inside other functions (local functions)
    a. Scope is the function

Regardless of where they're defined they follow the same form

- Keyword fun
- Function name
- Parameter list (optional)
    - Parameters are defined as 'val' and therefore cannot be changed in the function
- Return type (optional, can be inferred)
    - If a function does not return a value its return type is Unit (does not need to be specified)
- Function code in { }

```
fun sayHi(){
        println("Hello world")
}
sayHi()
```

Parameters must include their type. Multiple parameters are separated by a comma.
Parameters are defined as ‘val’ and therefore cannot be changed in the function.

```
fun sayHello(message: String){
        println(message)
}
sayHello("Hello class")
```

You can assign default values for parameters which will be used if no value is passed in for that parameter.
Most Compose libraries use default arguments as it makes composables customizable, but still makes the default behavior simple to invoke.

```
fun sayWhat(message: String = "I am the default message"){
        println(message)
}
sayWhat()
sayWhat("I am not the default messsage")
```

You can name parameters to improve code readability.

```
fun sayWho(message: String, name: String){
        println("$message $name")
}
sayWho("Hi there", "Honore")
sayWho("Hi there", name="Esther")
```

If a function does not return a value its return type is Unit

- Returned implicitly, no need for a return statement
- Unit is a class with only one instance/value
- Similar to void in Java

You can specify a return type after the list of parameters and a colon.

```
fun sayWhere(name: String, place: String): String{
        return "$name is going to $place"
}
println(sayWhere("Bienvenu", "Paris"))
```

When a function returns a single expression you can eliminate the curly braces and the keyword return and just assign the value as an expression.

```
fun sayWhereTo(name: String, place: String):String = "$name is going
to $place"
println(sayWhereTo("Sandrine", "Pittsburgh"))
```

The return type also becomes optional because it can be inferred in the expression.

```
fun sayWhereToNow(name: String, place: String) = "$name is going to
$place"
println(sayWhereToNow("Elie", "New York"))
```

Functions that are not declared but are passed as an expression are function literals.
In Kotlin lambda expressions and anonymous functions are function literals.

Lambda Expressions

A lambda expression is a self-contained block of code.
Lambda expressions are always wrapped in curly braces as we have in the example of the let function in our nullable example above. The syntax of Kotlin lambdas is similar to lambdas in Java.

- Parameters go inside the curly braces
- Parameter types can be omitted if they can be inferred
- The body goes after a ->
- If a lambda expression has only one parameter it does not need to be declared
    - the parameter will be implicitly declared as "it"
    - the -> can be omitted
- A return inside a lambda expression returns from the enclosing function

```
// with type annotation
val sum1 = {a: Int, b: Int -> a + b}
println(sum1(2, 3))

// without type annotation
val sum2:(Int,Int)-> Int  = {a , b -> a + b}
println(sum2(3, 4))
```

Lambdas have the same capabilities as many other data types. Particularly useful is that lambdas can be passed to functions as arguments and returned as results.

Lambda expressions don't have the ability to specify the functions return type. In most cases, this is unnecessary because the return type can be inferred automatically. However, if you do need to specify it explicitly, you can use an alternative syntax: an *anonymous function*.

Anonymous functions

The return type of a function literal can often be inferred but if you need to specify it you'll need an anonymous function. An anonymous function looks exactly like a regular function except its name is omitted.

- Parameter types can be omitted if they can be inferred
- A return inside an anonymous function returns from the anonymous function itself

Higher-order functions

A higher-order function is a function that takes functions as parameters or returns a function. Lambdas are very often used this way in Kotlin. This pattern is useful for communicating between components in the same way that you might use a callback interface in Java.

A function type is a combination of the parameter types it accepts and the type of result it returns. The parameter types are in parenthesis, separated by a comma. The return type is listed after an arrow, ->
When a higher-order function that has a parameter expecting a function is called you must pass in a lambda that matches the parameter's declared type.
(A, B) -> C
Parameters of types A and B and return a value of type C.

() -> A
No parameters, return value of type C

(A) -> Unit
Parameter of type A, nothing returned. The Unit return type cannot be omitted.

```
fun callMe(greeting: () -> Unit) {
   greeting()
}

callMe({ println("This is a lambda expression") })
```

`callMe()` is a higher-order function with the `greeting` parameter expecting a lambda expression as:

```
greeting: () -> Unit
```

The empty parenthesis mean that the lambda expression doesn't accept any parameters and the `Unit` keyword means no value is returned.

This higher-order function takes a lambda that with 2 parameters of type Int and returns an In.

```
fun sum(lambda:(Int, Int)->Int){

    var result = lambda(2, 4)

    println("The sum is $result")

}

sum({a: Int , b: Int -> a + b })
```

If the lambda expression being passed to a function is the only parameter you can omit the parenthesis.

```
sum{a: Int , b: Int -> a + b }
```

If the lambda expression being passed to a function is the last parameter you can pull it out of the parenthesis. Both of these situations and syntax are common in Compose.

If you'll be using a lambda expression multiple times you can assign it to a variable instead.

```
val l={a: Int , b: Int -> a + b }

sum(l)
```

Comments

Comments are used to include notes, citations, or explanations in your code.

- // is used for single line comments
- /* comment */ is used for multi-line comments