Developer Student Clubs

Kotlin Basics II

An introductory course to the Kotlin programming language

Stelios Papamichail
Android Engineer
@MikePapamichail

Google Developers

Agenda

- Lambda Functions
- If-else expressions
- Loops
- The when() expression
- Ranges
- Collections
- and finally nullable values!

Developer Student Clubs

Lambda (λ) Functions

Developer Student Clubs

Lambda Expressions

Lambda expressions (and anonymous functions) are *function literals*. Function literals are functions that are not declared, but are passed immediately as an expression. Consider the following example:

```
max(strings, { a, b -> a.length < b.length })
```

The function max is a *higher-order function*, as it takes a function value as its second argument. This second argument is an expression that is itself a function, called a function literal, which is equivalent to the following named function:

```
fun compare(a: String, b: String): Boolean = a.length < b.length
```

Lambda Expression Syntax

The full syntactic form of lambda expressions is as follows: `val sum: (Int, Int) → Int = { x: Int, y: Int → x + y }`

- A lambda expression is always surrounded by curly braces.
- Parameter declarations in the full syntactic form go inside curly braces and have optional type annotations.
- The body goes after the →.
- If the inferred return type of the lambda is not Unit, the last (or possibly single) expression inside the lambda body is treated as the return value.

If you leave all the optional annotations out, what's left looks like this:

```
val sum = { x: Int, y: Int → x + y }
```

Trailing Lambdas

According to Kotlin convention, if the last parameter of a function is a function, then a lambda expression passed as the corresponding argument can be placed outside the parentheses:

```
val product = items.fold(1) { acc, e → acc * e }
```

Such syntax is also known as ***trailing lambda***.
If the lambda is the only argument in that call, the parentheses can be omitted entirely:

```
run { println("...") }
```

The ‘it’ lambda parameter

It's very common for a lambda expression to have only one parameter.
If the compiler can parse the signature without any parameters, the parameter does not need to be declared and → can be omitted. The parameter will be implicitly declared under the name it:

```
ints.filter { it > 0 } // this literal is of type '(it: Int) → Boolean'
```

Lambda return values

You can explicitly return a value from the lambda using the qualified return syntax. Otherwise, the value of the last expression is implicitly returned.
Therefore, the two following snippets are equivalent:

```
ints.filter {
    val shouldFilter = it > 0
    shouldFilter
}
```

```
ints.filter {
    val shouldFilter = it > 0
    return@filter shouldFilter
}
```

Underscore for unused variables

If the lambda parameter is unused, you can place an underscore instead of its name:

```
map.forEach { (_, value) → println("$value!") }
```

Control Flow

Developer Student Clubs

If Expression

In Kotlin, **if is an expression**: it returns a value. Therefore, there is no ternary operator (condition ? then : else) because ordinary if works fine in this role. Example:

```
var max = a
if (a < b) max = b

// With else
var max: Int
if (a > b) {
   max = a
} else {
   max = b
}

// As expression
  val max = if (a > b) a else b
```

Branches of an if expression can be blocks. In this case, the last expression is the value of a block:

```
val max = if (a > b) {
   print("Choose a")
   a
} else {
   print("Choose b")
   b
  }
```

If you're using if as an expression, for example, for returning its value or assigning it to a variable, the else branch is mandatory.

The When Expression

when defines a conditional expression with multiple branches. It is similar to the switch statement in C-like languages. Its simple form looks like this.

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> {
        print("x is neither 1 nor 2")
    }
}
```

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 first matching 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.

The When Expression

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, for example, with enum class entries and sealed class subtypes.

```
enum class Bit {
 ZERO, ONE
}


val numericValue = when (getRandomBit()) {
   Bit.ZERO -> 0
   Bit.ONE -> 1
   // 'else' is not required because all cases are covered
}
```

For Loops

The for loop iterates through anything that provides an iterator. This is equivalent to the foreach loop in languages like C#. The syntax of for is the following:

```
for (item in collection) print(item)
```

The body of for can be a block.

```
for (item: Int in ints) {
   // ...
}
```

We will see more example of the for loop later on, when we will cover ranges!

While Loops

While and do-while loops execute their body continuously while their condition is satisfied. The difference between them is the condition checking time:

- while checks the condition and, if it's satisfied, executes the body and then returns to the condition check.
- do-while executes the body and then checks the condition. If it's satisfied, the loop repeats. So, the body of do-while executes at least once regardless of the condition.

```
while (x > 0) {
  x--
}

do {
  val y = retrieveData()
} while (y != null) // y is visible here!
```

Return & Jumps

Kotlin has three structural jump expressions:

- return by default returns from the nearest enclosing function or anonymous function.
- break terminates the nearest enclosing loop.
- continue proceeds to the next step of the nearest enclosing loop.

All of these expressions can be used as part of larger expressions:

```
val s = person.name ?: return
```

The type of these expressions is the Nothing type.

Kotlin also supports labels for break, continue & return statements! Find more here.

Classes & Objects

Developer Student Clubs

Classes

Classes in Kotlin are declared using the keyword class:

```
class Person { /*...*/ }
```

The class declaration consists of the class name, the class header (specifying its type parameters, the primary constructor, and some other things), and the class body surrounded by curly braces. Both the header and the body are optional; if the class has no body, the curly braces can be omitted.

```
class Empty
```

Classes

A class in Kotlin can have a *primary constructor* and one or more *secondary constructors*. The primary constructor is a part of the class header, and it goes after the class name and optional type parameters.

```
class Person constructor(firstName: String) { /*...*/ }
```

If the primary constructor does not have any annotations or visibility modifiers, the constructor keyword can be omitted:

```
class Person(firstName: String) { /*...*/ }
```

The primary constructor cannot contain any code. Initialization code can be placed in *initializer blocks* prefixed with the init keyword.

Classes

During the initialization of an instance, the initializer blocks are executed in the same order as they appear in the class body, interleaved with the property initializers:

```
class InitOrderDemo(name: String) {
  val firstProperty = "First property: $name".also(::println)

  init {
     println("First initializer block that prints $name")
  }

  val secondProperty = "Second property: ${name.length}".also(::println)

  init {
     println("Second initializer block that prints ${name.length}")
  }
}
```

Classes

Kotlin has a concise syntax for declaring properties and initializing them from the primary constructor:

```kotlin
class Person(val firstName: String, val lastName: String, var age: Int)
```

Such declarations can also include default values of the class properties:

```kotlin
class Person(val firstName: String, val lastName: String, var isEmployed: Boolean = true)
```

Much like regular properties, properties declared in the primary constructor can be mutable (var) or read-only (val).

If the constructor has annotations or visibility modifiers, the constructor keyword is required and the modifiers go before it:

```kotlin
class Customer public @Inject constructor(name: String) { /*...*/ }
```

Instantiating Classes

To create an instance of a class, call the constructor as if it were a regular function:

```
val invoice = Invoice()

val customer = Customer("Joe Smith")
```

Kotlin does not have a new keyword.

The process of creating instances of nested, inner, and anonymous inner classes is described in Nested classes.

Class Members

Classes can contain:

- Constructors and initializer blocks
- Functions
- Properties
- Nested and inner classes
- Object declarations

There is a myriad of things left to cover about classes, such as Inheritance, Abstract classes, Interfaces, Sealed classes, In/Out Classes & much more! Some of those will be covered in our next event, but you are strongly encouraged to skim through the documentation!

Kotlin Ranges

Ranges

Kotlin lets you easily create ranges of values using the rangeTo() function from the kotlin.ranges package and its operator form .. Usually, rangeTo() is complemented by in or !in functions.

```
if (i in 1..4) { // equivalent of i >= 1 && i <= 4
  print(i)
}
```

Integral type ranges (IntRange, LongRange, CharRange) have an extra feature: they can be iterated over. These ranges are also progressions of the corresponding integral types.

Range definition

A range defines a closed interval in the mathematical sense: **it is defined by its two endpoint values which are both included in the range**. Ranges are defined for comparable types: having an order, you can define whether an arbitrary instance is in the range between two given instances.

The main operation on ranges is **contains**, which is usually used in the form of in and !in operators.

To create a range for your class, call the *rangeTo()* function on the range start value and provide the end value as an argument. *rangeTo()* is often called in its operator form ..

```
val versionRange = Version(1, 11)..Version(1, 30)
println(Version(0, 9) in versionRange)
println(Version(1, 20) in versionRange)
```

Iterating using ranges

Ranges are generally used for iteration in for loops.

```
for (i in 1..4) print(i)
```

To iterate numbers in reverse order, use the downTo function instead of the... operator:

```
for (i in 4 downTo 1) print(i)
```

It is also possible to iterate over numbers with an arbitrary step (not necessarily 1). This is done via the step function:

```
for (i in 1..8 step 2) print(i)

println()

for (i in 8 downTo 1 step 2) print(i)
```

To iterate a number range which does not include its end element, use the until function:

```
for (i in 1 until 10) {      // i in 1 until 10, excluding 10

  print(i)

}
```

Kotlin Collections

Developer Student Clubs

Collections Overview

The Kotlin Standard Library provides a comprehensive set of tools for managing *collections* – groups of a variable number of items (possibly zero) that are significant to the problem being solved and are commonly operated on.

It also provides implementations for basic collection types: sets, lists, and maps. A pair of interfaces represent each collection type:

- A *read-only* interface that provides operations for accessing collection elements.
- A *mutable* interface that extends the corresponding read-only interface with write operations: adding, removing, and updating its elements.

List<T> Collection

- List<T> stores elements in a specified order and provides indexed access to them. Indices start from zero – the index of the first element – and go to lastIndex which is the (list.size - 1). For example: *val numbers = listOf("one", "two", "three", "four")*
- MutableList<T> is a List with list-specific write operations, for example, to add or remove an element at a specific position.

```
val numbers = mutableListOf(1, 2, 3, 4)
numbers.add(5)
numbers.removeAt(1)
numbers[0] = 0
numbers.shuffle()
println(numbers)
```

As you see, in some aspects, lists are very similar to arrays. However, there is **one important difference**: an array's size is defined upon initialization and is never changed; in turn, a list doesn't have a predefined size; a list's size can be changed as a result of write operations: adding, updating, or removing elements.

In Kotlin, the default implementation of List is ArrayList which you can think of as a resizable array.

Set<T> Collection

Set<T> stores unique elements; their order is generally undefined. null elements are unique as well: a Set can contain only one null. Two sets are equal if they have the same size, and for each element of a set there is an equal element in the other set.

```
val numbers = setOf(1, 2, 3, 4)
println("Number of elements: ${numbers.size}")
if (numbers.contains(1)) println("1 is in the set")

val numbersBackwards = setOf(4, 3, 2, 1)
println("The sets are equal: ${numbers == numbersBackwards}")
```

MutableSet<T> Collection

MutableSet is a Set with write operations from MutableCollection.
The default implementation of Set – LinkedHashSet – preserves the order of elements insertion. Hence, the functions that rely on the order, such as first() or last(), return predictable results on such sets.

```kotlin
val numbers = setOf(1, 2, 3, 4)  // LinkedHashSet is the default implementation
val numbersBackwards = setOf(4, 3, 2, 1)
println(numbers.first() == numbersBackwards.first())
println(numbers.first() == numbersBackwards.last())
```

An alternative implementation – HashSet – says nothing about the elements order, so calling such functions on it returns unpredictable results. However, HashSet requires less memory to store the same number of elements.

Map<K,V> Collection

Map<K, V> is not an inheritor of the Collection interface; however, it's a Kotlin collection type as well. A Map stores *key-value* pairs (or *entries*); keys are unique, but different keys can be paired with equal values. The Map interface provides specific functions, such as access to value by key, searching keys and values, and so on.

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
println("All keys: ${numbersMap.keys}")
println("All values: ${numbersMap.values}")
if ("key2" in numbersMap) println("Value by key \"key2\": ${numbersMap["key2"]}")
if (1 in numbersMap.values) println("The value 1 is in the map")
if (numbersMap.containsValue(1)) println("The value 1 is in the map") // same as previous
```

MutableMap<K,V> Collection

MutableMap is a Map with map write operations, for example, you can add a new key-value pair or update the value associated with the given key.

```kotlin
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
numbersMap.put("three", 3)
numbersMap["one"] = 11
println(numbersMap)
```

The default implementation of Map – LinkedHashMap – preserves the order of elements insertion when iterating the map. In turn, an alternative implementation – HashMap – says nothing about the elements order.

Null Safety!!

Developer Student Clubs

Nullable & non-nullable types

Kotlin's type system is aimed at **eliminating the danger of null references**, also known as The Billion Dollar Mistake.

One of the most common pitfalls in many programming languages, including Java, is that accessing a member of a null reference will result in a null reference exception. In Java, this would be the equivalent of a NullPointerException, or an *NPE* for short.
The only possible causes of an NPE in Kotlin are:

- An explicit call to throw NullPointerException().
- Usage of the !! operator that is described later.
- Data inconsistency with regard to initialization, such as when:
  - An uninitialized this available in a constructor is passed and used somewhere (a "leaking this").
  - A superclass constructor calls an open member whose implementation in the derived class uses an uninitialized state.
- Java interoperation:
  - Attempts to access a member of a null reference of a platform type;
  - Nullability issues with generic types being used for Java interoperation. For example, a piece of Java code might add null into a Kotlin MutableList<String>, therefore requiring a MutableList<String?> for working with it.
  - Other issues caused by external Java code.

Nullable & non-nullable types

In Kotlin, the type system distinguishes between references that can hold null (**nullable** references) and those that cannot (**non-null** references).
For example, a regular variable of type String cannot hold null:

```
var a: String = "abc" // Regular initialization means non-null by default

a = null // compilation error
```

To allow nulls, you can declare a variable as a nullable string by writing String?:

```
var b: String? = "abc" // can be set to null

b = null // ok

print(b)
```

Now, if you call a method or access a property on a, it's guaranteed not to cause an NPE, so you can safely say:

```
val l = a.length
```

But if you want to access the same property on b, that would not be safe, and the compiler reports an error:

```
val l = b.length // error: variable 'b' can be null
```

But you still need to access that property, right? Let's see how to achieve this!

Checking for null conditions

First, you can explicitly check whether b is null, and handle the two options separately:

```
val l = if (b != null) b.length else -1
```

The *compiler tracks the information* about the check you performed, and allows the call to length inside the if. More complex conditions are supported as well:

```
val b: String? = "Kotlin"
if (b != null && b.length > 0) {
  print("String of length ${b.length}")
} else {
  print("Empty string")
}
```

Note that **this only works where b is immutable** (meaning it is a local variable that is not modified between the check and its usage or it is a member val that has a backing field and is not overridable), because otherwise it could be the case that b changes to null after the check.

Safe calls

Your second option for accessing a property on a nullable variable is using the safe call operator ?. :

```
val a = "Kotlin"

val b: String? = null

println(b?.length)

println(a?.length) // Unnecessary safe call
```

This returns b.length if b is not null, and null otherwise. The type of this expression is Int?.

Safe calls are useful in chains. For example, Bob is an employee who may be assigned to a department (or not). That department may in turn have another employee as a department head. To obtain the name of Bob's department head (if there is one), you write the following:

```
bob?.department?.head?.name
```

Such a chain returns null if **any of the properties** in it is null.

Safe calls

To perform a certain operation **only for non-null values**, you can use the safe call operator together with let:

```
val listWithNulls: List<String?> = listOf("Kotlin", null)
for (item in listWithNulls) {
  item?.let { println(it) } // prints Kotlin and ignores null
}
```

A safe call can also be placed on the left side of an assignment. Then, if one of the receivers in the safe calls chain is null, the assignment is skipped and the expression on the right is not evaluated at all:

```
// If either `person` or `person.department` is null, the function is not called:
person?.department?.head = managersPool.getManager()
```

The elvis operator

When you have a nullable reference, b, you can say "if b is not null, use it, otherwise use some non-null value":

```
val l: Int = if (b != null) b.length else -1
```

Instead of writing the complete if expression, you can also express this with the Elvis operator ?: :

```
val l = b?.length ?: -1
```

If the expression to the left of ?: is not null, the Elvis operator returns it, otherwise it returns the expression to the right. Note that the expression on the right-hand side is evaluated only if the left-hand side is null.

Since throw and return are expressions in Kotlin, they can also be used on the right-hand side of the Elvis operator. This can be handy, for example, when checking function arguments:

```
fun foo(node: Node): String? {
    val parent = node.getParent() ?: return null
    val name = node.getName() ?: throw IllegalArgumentException("name expected")
    // ...
}
```

The !! operator

The third option is for NPE-lovers: the not-null assertion operator (!!) converts any value to a non-null type and throws an exception if the value is null. You can write b!!, and this will return a non-null value of b (for example, a String in our example) or throw an NPE if b is null:

```
val l = b!!.length
```

Thus, if you want an NPE, you can have it, but you have to ask for it explicitly, and it won't appear out of the blue.

Don’t say we didn’t warn you ;)

Safe Casts

Regular casts may result in a ClassCastException if the object is not of the target type. Another option is to use safe casts that return null if the attempt was not successful:

```
val aInt: Int? = a as? Int
```

Closing Words

Thank you for your time, everybody!

We hope that you learned something new today. There's a TON of awesome things that we couldn't cover today, so if you're interested in diving a bit deeper, here are a few resources:

- Kotlin Docs
- Kotlin Playground
- Kotlin by example
- Kotlin Hands-On

Special thanks to the @GoogleDevs that are making initiatives such as this possible! Don't forget to **follow the GDSC UoC on our social media & join our discord server** to stay up to date with upcoming events!

Thank you for your time!

Developer Student Clubs