Getting Started + Basics + Functions and Lambdas

2020/12/01

Useful Tool

[IntelliJ] Decompile from Kotlin to Java code
https://ijeee.tistory.com/22

The formatting shortcuts in Intellij IDEA are
For Windows : Ctrl + Alt + L.
For Ubuntu : Ctrl + Alt + Windows + L.
For Mac : ⌥ (Option) + ⌘ (Command) + L.

Package and Import

```
package org.example

fun printMessage() { /*...*/ }
class Message { /*...*/ }

// ...
```

```
import org.example.Message
import org.test.Message as testMessage
```

```
import org.example.*
```

Program Enty Point

Koltin

```
fun main(): Unit {
    // ...
}
```

Java

```
public class TestClass {
    public static void main(String[] args) {
        // ...
    }
}
```

Basic Types

- In Kotlin, everything is an object
- So we can call member functions and properties on any variable.

```kotlin
/**
 * Represents a 32-bit signed integer.
 * On the JVM, non-nullable values of this type are represented as values of the primitive type `int`.
 */
public class Int private constructor() : Number(), Comparable<Int> {
    companion object {
        /**
         * A constant holding the minimum value an instance of Int can have.
         */
        public const val MIN_VALUE: Int = -2147483648

        /**
```

Basic Types

```
val one = 1 // Int
val threeBillion = 3000000000 // Long
val oneLong = 1L // Long
val oneByte: Byte = 1
```

- (basically) Int

---> (if value is exceedes maximum value or with explicit expression) Long

```
val pi = 3.14 // Double
val e = 2.7182818284 // Double
val eFloat = 2.7182818284f // Float, actual value is 2.7182817
```

- (basically) Double

---> (with explicit expression) Float

Basic Types

```
val oneMillion = 1_000_000
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

_ makes number more readable!

Basic Types

- No implicit widening conversions for numbers in Kotlin.

```
fun main() {
    fun printDouble(d: Double) { print(d) }

    val i = 1
    val d = 1.1
    val f = 1.1f

    printDouble(d)
//    printDouble(i) // Error: Type mismatch
//    printDouble(f) // Error: Type mismatch
}
```

Basic Types

- No implicit widening conversions for numbers in Kotlin.
- smaller types are NOT implicitly converted to bigger types.

```
fun main() {
    fun printDouble(d: Double) { print(d) }

    val i = 1
    val d = 1.1
    val f = 1.1f

    printDouble(d)
    printDouble(i.toDouble())
    printDouble(d.toDouble())
}
```

Basic Types

```
val str = "abcde"
for (c in str) {
    println(c)
}
for(i in 0..str.length-1) {
    println(str[i])
}
```

```
val a = 100
println("a: $a")
println("a.plus(1): ${a.plus( other: 1)}")
```

```
val price = """
${'$'}9.99
"""
```

```
$9.99
```

Basic Types

```
val text = """
>>Tell me and I forget.
>>Teach me and I remember.
>>Involve me and I learn.
>>(Benjamin Franklin)
""".trimMargin()
```

```
    >>Tell me and I forget.
    >>Teach me and I remember.
    >>Involve me and I learn.
    >>(Benjamin Franklin)

BUILD SUCCESSFUL in 370ms
```

```
val text = """
>>Tell me and I forget.
>>Teach me and I remember.
>>Involve me and I learn.
>>(Benjamin Franklin)
""".trimMargin( marginPrefix: ">")
```

```
>Tell me and I forget.
>Teach me and I remember.
>Involve me and I learn.
>(Benjamin Franklin)

BUILD SUCCESSFUL in 397ms
```

```
val text = """
>>Tell me and I forget.
>>Teach me and I remember.
>>Involve me and I learn.
>>(Benjamin Franklin)
""".trimMargin( marginPrefix: ">>")
```

```
Tell me and I forget.
Teach me and I remember.
Involve me and I learn.
(Benjamin Franklin)

BUILD SUCCESSFUL in 355ms
```

Nullable values and null checks

```
val x: Int = 10
val y: Int? = 20
```

- A reference must be explicitly marked as nullable when null value is possible.

Type checks and automatic casts

```
val str:String? = "abcde"
str.toDouble()

if(

}
```

Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?

Surround with null check ⌥⇧⏎ More actions... ⌥⏎

val str: String?

```
if(str is String) {
    str.toDouble()
}
```

Variable

val >> getter, ~~setter~~
var >> getter, setter

```
class B {
    val a: Int = 10
    var b: Int = 0
}
```

```
public final class B {
   private final int a = 10;
   private int b;

   public final int getA() {
      return this.a;
   }

   public final int getB() { ... }

   public final void setB(int var1) {
      this.b = var1;
   }
}
```

Control Flow: If

- In Kotlin, if expression returns a value

```
// Traditional usage
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
```

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

Control Flow: When

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

```
when {
    x.isOdd() -> print("x is odd")
    y.isEven() -> print("y is even")
    else -> print("x+y is even.")
}
```

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

```
fun setValue(x: Int) : Unit = when(x) {
    1 -> println("x is 1")
    else -> println("x is not 1")
}
```

Control Flow: For

```
for (i in 1..3) {
    println(i)
}
```

```
val range1 = 1..10
~~val range2 = 10..1 // ???~~
val range3 = 10 downTo 1
val range4 = 10 downTo 1 step 2
```

```
val array = arrayOf("a", 'b')
for ((index, value) in array.withIndex()) {
    println("the element at $index is $value")
}
// the element at 0 is a
// the element at 1 is b
```

Control Flow: While

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

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

Returns and Jumps

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

Return type is Nothing type

Returns and Jumps

- 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

```
for (i in 1..3) {
    for (j in 1..3) {
        println("i: $i, j: $
        if (i == 2) break
    }
}
```

```
i: 1, j: 1
i: 1, j: 2
i: 1, j: 3
i: 2, j: 1
i: 3, j: 1
i: 3, j: 2
i: 3, j: 3
```

```
for (i in 1..3) {
    for (j in 1..3) {
        println("i: $i,
        if (i == 2) cont
    }
}
```

```
i: 1, j: 1
i: 1, j: 2
i: 1, j: 3
i: 2, j: 1
i: 2, j: 2
i: 2, j: 3
i: 3, j: 1
i: 3, j: 2
i: 3, j: 3
```

Returns and Jumps

```
loop@ for (i in 1..3) {
    for (j in 1..3) {
        println("i: $i, j: $j")
        if (i == 2) break@loop
    }
}
```

```
i: 1, j: 1
i: 1, j: 2
i: 1, j: 3
i: 2, j: 1
```

```
loop@ for (i in 1..3) {
    for (j in 1..3) {
        println("i: $i, j: $j")
        if (i == 2) continue@loop
    }
}
```

```
i: 1, j: 1
i: 1, j: 2
i: 1, j: 3
i: 2, j: 1
i: 3, j: 1
i: 3, j: 2
i: 3, j: 3
```

Returns and Jumps

```
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach {
        if (it == 3) return
        print(it)
    }
    println("this point is unreachable")
}
```

```
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach lit@{
        if (it == 3) return@lit
        print(it)
    }
    print(" done with explicit label")
}
```

Returns and Jumps

```
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach {
        if (it == 3) return@forEach
        print(it)
    }
    print(" done with implicit label")
}
```

```
fun foo() {
    run loop@{
        listOf(1, 2, 3, 4, 5).forEach {
            if (it == 3) return@loop
            print(it)
        }
    }
    print(" done with nested loop")
}
```

Returns and Jumps

```
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach(fun(value: Int) {
        if (value == 3) return
        print(value)
    })
    print(" done with anonymous function")
}
```

```
return@a 1
```

return 1 at label @a

Nothing Type ?

```
val s = person.name ?: throw IllegalArgumentException("Name required")
```

```
fun fail(message: String): Nothing {
    throw IllegalArgumentException(message)
}
```

Throw expression: special type of **Nothing**

- No values
- It is used to mark code locations that can naver be reached

Nothing Type ?

```
val x: Nothing? = null              // 'x' has type `Nothing?`
val l: List<Nothing?> = listOf(null)
```

Nullable variant of Nothing (Nothing?) -> null

Functions

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

```
fun sum(a: Int, b: Int) = a + b
```

```
fun sum(
        a: Int,
        b: Int,
) = a + b
```

```
val rv1: Int = sum(10, 20)
val rv2:Int = sum(a=10,
b=20)
val rv3 = sum(b=20, a=10)
```

```
fun main(): Unit {
    // ...
}
```

```
fun main() {
    // ...
}
```

Functions – Default arguments

```
fun foo(
    bar: Int = 0,
    baz: Int,
) { /*...*/ }
```

```
foo(baz = 1)
// The default value bar = 0 is used
```

```
fun foo(
    bar: Int = 0,
    baz: Int = 1,
    qux: () -> Unit,
) { /*...*/ }
```

```
foo(1) { println("hello") }
// Uses the default value baz = 1

foo(qux = { println("hello") })
// Uses both default values bar = 0 and baz = 1

foo { println("hello") }
// Uses both default values bar = 0 and baz = 1
```

Functions – variable number of arguments

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts is an Array
        result.add(t)
    return result
}
```

```
val list = asList(1, 2, 3)
```

```
val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)
```

spread operator

Functions – Infix notation

```
infix fun Int.add(a1: Int): Int {
    return this + a1
}
```

```
val r1 = 100 add 50
// called Int.add(), and in the function this = 100
val r2 = 100.add(50)
```

```
class MyStringCollection {
    infix fun add(s: String) { /*...*/ }

    fun build() {
        this add "abc"   // Correct
        add("abc")       // Correct
        //add "abc"        // Incorrect: the receiver must be specified
    }
}
```

Lambdas

```
val lambda: (Int, Int) -> Int = {
        a1: Int, a2: Int -> a1 + a2
}
val lambda2 = {
        a1: Int, a2: Int -> a1 + a2
}
val lambda3: (Int, Int) -> Int = {
        a1, a2 -> a1 + a2
}
```

```
// Function vs Lambda

fun fun1 (a1: Int, a2: Int): Int {
    val r1 = a1 + a2
    val r2 = a1 - a2
    val r3 = r1 * r2
    return r3
}
val lambda4 = {
    a1: Int, a2: Int ->
    val r1 = a1 + a2
    val r2 = a1 - a2
    r1 * r2
}
```

Inline Functions

```
fun nonInlineFun() {
    println("----------------------")
    println("what is inline function")
    println("----------------------")
}
```

```
fun fun_nonInline() {
    nonInlineFun()
}
```

```
inline fun inlineFun() {
    println("----------------------")
    println("what is inline function")
    println("----------------------")
}
```

```
fun fun_inline() {
    inlineFun()
}
```

Inline Functions

```
public static final void fun38_nonInline() {
    nonInlineFun();
}
```

```
public static final void fun38_inline() {
    int $i$f$inlineFun = false;
    String var1 = "----------------------";
    boolean var2 = false;
    System.out.println(var1);
    var1 = "what is inline function";
    var2 = false;
    System.out.println(var1);
    var1 = "----------------------";
    var2 = false;
    System.out.println(var1);
}
```

Thank you!

Questions

- Nothing vs Null
- Nothing - throw exception?

-> https://agrawalsuneet.github.io/blogs/difference-between-any-unit-and-nothing-kotlin/

- In lambda, Can it has return keyword?

-> https://stackoverflow.com/q/45348820

- Example using ‘return@a 1’ ?

```
fun sum(a: Int, b:Int, lambdaFun:(Int, Int)->Int): Int {
    return lambdaFun(a, b)
}
fun main() {
    sum( a: 10,  b: 20) {a:Int, b:Int ->
        if(a == 1) return@sum 1
        a + b
    }
}
```