KOTLIN

Created: 2021-02-05 Fr 10:16

1 GENERAL

- cross-platform, general-purpose
- interoperable with Java
- statically-typed language by JetBrains
- main developing language for Android
- Documentation: https://kotlinlang.org/docs/reference/

2 HISTORY

- development started in 2010
- probably named after a Russian island
- 2011 announced by JetBrains
- 2017 Kotlin support in Android

3 DESIGN GOALS

- open-source (Apache License 2.0)
- interoperability
- concise and expressive
- type-safety
- adds functional programming styles
- data classes

4 SETUP

start by using IntelliJ https://www.jetbrains.com/idea/

`public static void main() {}` ⇒ `fun main() {}`

5 VARIABLES

- immutable values -> `val name: String = "note"` gives error upon re-assignment
- mutable variables -> `var`
- types are non-null by default
- nullable with ? However, this requires to check for null, the compiler will go mad about this otherwise

```
var name: String? = "test"
name?.trim()?.get(3)
```

5.1 STRING INTERPOLATION

```
val name = "Carsten"
println("Hello $name")
```

5.2 COLLECTIONS

```
val array = arrayOf("a", "b", "c")
val list = listOf("a", "b", "c")
val map = mapOf(0 to "a", 3 to "d")

val oArray = Array<Int>(6) { it -> it*2 }
```

A map is actually a list of pairs. There is a convenience function to create new pairs with `to`.

6 CONTROL FLOW

6.1 LOOPS

```
if (array[0] == a) {
    // is true
}
if (list.size == 4) {
    // is false
}
for (element in list) {
    println(element)
}
```

6.1.1 FUNCTIONAL STYLE

```
list.forEach { println(it) }
array.forEach { otherName -> println(otherName) }
val oArray = Array<Int>(6) { it -> it*2 }
```

Trailing lambda function arguments can be omitted and provided in curly braces after the actual function call. If the lambda function is the only argument, then you may even omit the normal braces for the function call (as in the example above).

```
list.forEachIndexed { index, element ->
    println("$element is at index $index")
}
map.forEach { key, value -> println("key: $key, val: $value") }
```

As before, all collections are immutable by default. If we want to have a mutable collection, we can eg. use a `mutableListOf`.

6.2 WHEN TO SWITCH

Instead of Java's switch case, kotlin has when expressions.

```
when (name) {
    null -> println("bla")
    else -> println("else")
}
```

If clauses are expressions, and, thus, can be used in assignments

```
val nameToPrint = if (name != null) name else "dummy"
val shortPrintName = name ?: "dummy"
```

The same is true for when statements

7 FUNCTIONS

7.1 GENERAL DEFINITON

```kotlin
fun getString(input: Int = 3): String {
    return input.toString()
}
```

Default values for an argument can be supplied.
No return value (java-void) is `Unit` in kotlin, it's the default return type of a function.
Functions do not need to be part of a class. Remember, java only knows functions as class members.

7.2 SINGLE EXPRESSION FUNCTIONS

```
fun getString(input: Int = 3) = input.toString()
fun greeting(msg: String, to: String) = println("$msg $to")
```

return value is inferred by the compiler's type inference

7.3 NAMED ARGUMENTS

```
fun greet(msg: String, to: String) = println("$msg $to")

fun main() {
    greet("Hi", "Carsten")
    greet(to = "Carsten", msg = "Hi")
}
```

8 CLASSES

8.1 IMPLICIT CONSTRUCTOR

```
class Person
//Implicit constructor

val p = Person()
```

8.2 CONSTRUCTORS

```
class Person(_firstName: String, _lastName: String) {
    val firstName: String
    val lastName = _lastName
    init {
        //always run a new instance is created
        // can appear multiple times, run in order
        firstName = _firstName
    }
}

val p = Person("Peter", "Schneider")
println(p.firstName)
```

8.2.1 DEFAULTS AND PROPERTIES

```
class Person(val firstName: String, val lastName: String) {
    constructor(): this("Unknown", "Person")
    // secondary constructor run after init blocks
}

val p = Person("Peter", "Schneider")
println(p.firstName)

class Person(val firstName: String = "Peter", val lastName: String = "Schneider")

val p = Person()
println(p.firstName)
```

8.3 PROPERTIES

```
class Person(val name: String = "Peter") {
    var nickname: String? = null
        set(value) {
            field = value
            println(value)
        }
        get() {
            println("Giving you $field")
            return field
        }
}
```

8.4 METHODS ARE FUNCTIONS INSIDE CLASSES

```
class Person(val name: String = "Peter") {
    var nick: String? = null
    fun info() {
        //val nickToPrint = if (nick != null) nick else "no nickname"
        // use elvis operator -> ?:
        val nickToPrint = nick ?: "no nickname"
        println("$name ($nick)")
    }
}

val p = Person()
p.info()
```

8.5 THERE IS MORE

- eg. interfaces

```
interface YourPersonInterface {
    fun printInfo(name: String)
}
class Person : YourPersonInterface
```

- data classes

```
data class Person(val name: String, var nick: String)

val p = Person("Peter", "nonick")
println(p)
```

- enum classes

```
enum class PersonType {
    Natural, Artificial
}
val myType = PersonType.Natural
```

9 GRADLE

```
plugins {
    // Apply the org.jetbrains.kotlin.jvm Plugin to add support for Kotlin.
    id("org.jetbrains.kotlin.jvm") version "1.4.20"
    id("org.jetbrains.kotlin.plugin.serialization") version "1.4.30-RC"
    // Apply the application plugin to add support for building a CLI application in Java.
    application
}
repositories {
    // Use JCenter for resolving dependencies.
    jcenter()
}
dependencies {
    // Align versions of all Kotlin components
    implementation(platform("org.jetbrains.kotlin:kotlin-bom"))
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-cbor:1.0.1")
}
application {
    // Define the main class for the application.
    mainClass.set("security2.kt.AppKt")
}
```

- `gradle init` ⇒ create a new project
- `gradle check` ⇒ build + unit tests
- `gradle run` ⇒ build + run main class