Kotlin Ecosystems

Author: Daria Shutina

23-02-01

Org stuff

Homeworks

If homework is given at date DATE, the soft deadline is DATE + 2 weeks and the hard deadline is at DATE + 3 weeks.

Some homeworks will be connected to each other, so it is bad to skip one of them.

Homeworks will appear after lectures. `git push` to a separate branch, then open a PR.

Grade

To pass the course you have to score ≥ 45 points.

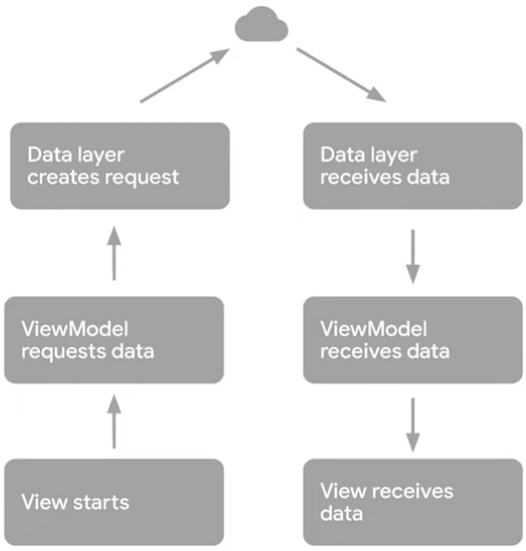
To get A+ you have to score ≥ 95 points.

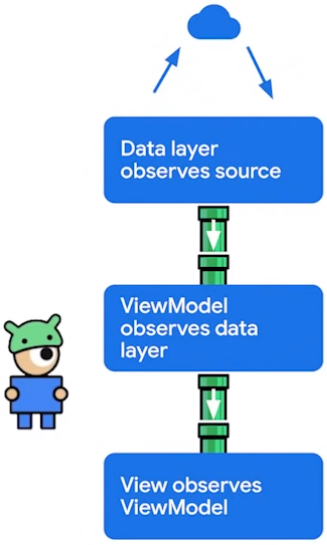
There are 130 points you can score in this course: 74 for home assignments, 44 for quizzes, 12 for the test.

Cold flows

Main idea

Ordinary threads work in the way that, first, the user sends the request, then they get the answer from the server. Kotlin flows simplify this scheme for the user. Data is "preloaded" to the user.





This type of flows is called "cold", because they are created on demand and emit data only when they are being observed.

Flows should be used in a Coroutine Scope.

Flow builder

```
class UserMessageDataSource(
    private val messageApi: MessagesApi,
    private val refreshIntervalMs: Long = 5000
) {
    val latestMessages: Flow<List<Message>> = flow {
        while(true) { // in order to infinitely fetch messages
            val userMessages = messageApi.fetchLatestMessages()
            emit(userMessages) // emit the result to the flow
            delay(refreshIntervalMs) // wait
        }
    }
}
```

The code inside `while(true)` cycle is called *the producer block*.

Modifying flows

`flow.map`, `flow.filter` and `flow.catch`

```
val userMessages: Flow<MessagesUiModel> =
    UserMessageDataSource.latestMessages
        .map { it. toUiModel() }
        .filter { it.containsImportantNotifications() }
        .catch { e ->
            analytics.log("Error loading reserved event")
            if (e is IllegalArgumentException) throw e
            else emit(emptyList())
        }
```

Observing flows

`flow.collect`

```
userMessages.collect { listAdapter.submitList(it) }
```

Every time `flow.collect` is called on `userMessages`, a new flow will be created. And its producer block will start refreshing messages from the API at its own interval (ничево не понятно)

23-02-08

Gradle

Settings

If there are several modules in a project, each module should have its own settings. The settings are saved in a directory `build.gradle.hts`.

Data in `settings.gradle.kts` will be executed before other settings.

To add a module, use `Project -> New -> Module`.

Tasks

There are default and custom tasks, tasks from plugins. Custom tasks can be defined in the build configuration.

```
task.register("name") {
    group = "useless"
    dependsOn(tasks.named("othername"))
    println("${this.name}, configuration")
    doFirst {
        println("${this.name}, first in execution")
    }
}

task.register("othername") {
    println("${this.name}, configuration")
    doFirst {
        println("${this.name}, first in execution")
    }
    doLast {
        println("${this.name}, last in execution")
    }
}
```

`dependsOn` means that tasks with the name `othername` will be executed before the `name` tasks.

You can also play with tasks:

```
tasks
    .filter { task ->
        task.group?. let { it == "useless" } ?: false
    }
    .forEach {
        it.dependsOn(task3)
    }
```

Throwback

`./gradlew build` will use all components. Use a flag `-x test` to exclude tests, for example.

Plugins

Most useful features are added by plugins.

There are binary plugins and script plugins. If you set up a plugin correctly, you will see additional tasks.

Applying a community plugin:

```
plugins {
    kotlin("jvm") version "1.8.0"
    id("io.gitlab.arturbosch.detekt") version "1.21.0" apply false
}
```

`apply false` means the plugin will not be added in the beginning of the project build. By default, the value is `true`.

23-02-15

Exceptions

Errors were basically functions that return special error values. Thus, they signaled that something bad has happened. Another way was to set some global variable (i.e. `errno`). You have to handle errors even if you do not care about them.

Exceptions stop the program ad go to the top of the call stack.

Do not use exceptions for control flows.

Usage of `throw`

```
fun main() {
    try {
        throw Exception("An exception", RuntimeException("A cause"))
    } catch (e: Exception) {
        println("Message: ${e.message}")
        println("Cause: ${e.cause}")
        println("Exception: $e")
        e.printStackTrace()
    } finally {
        println("Finally always executes")
    }
}
```

Under the JVM hood

An exception is thrown. JVM stop the work and looks for smth which is called an exception table. If it does not find an appropriate exception in the exception table, it levels up and looks for another exception table.

Finally, if JVM does not find anything in exception tables, it goes to exception handler.

Custom exceptions

```
class MyException(val op: String, vararg val args: Int)
    : RuntimeException("Problem with $op for args $args")

fun test() {
    try {
        doSomething()
    } catch (e: MyException) {
        println("Maybe retry with ${e.op}?")
    } catch (e: Error) {
        println("Error happened! $err")
    } catch (e: Throwable) {
        println("Wtf?")
        throw t
    }
}
```

(Un)checked exceptions

In Java exceptions can be unchecked (Error/RuntimeException) and checked (the rest).

In Kotlin all exceptions are unchecked.

Testing \TODO

Principles

1. Testing demonstrates the presence of defects, but it does not prove that there are none of them.
2. Testing involves concrete program executions
3. The earlier the better. The cost of repair grows exponentially.

How to find a bug

1. Run the software
2. Run the reference model
3. Compare the results

Unit testing in Kotlin

Each non-trivial function has its own block of tests. The tests should run fast and one test should check a small part of the model.

```
// build.gradle.kts

dependencies {
    ...
    testImplementation(platform("org.junit:junit-bom:5.8.2"))
    testImplementation("org.junit.jupiter:junit-jupiter:5.8.2")
}

tasks.test {
    useJUnitPlatform()
}
```

Example: kotlin program

```kotlin
class MyTests {
    @Test
    @Tag("main")
    @DisplayName("Check if the calculator works correctly")
    fun testCalculatr() {
        Assertions.assertEquales(
            3,
            myCalculator(1, 2, "+"),
            "Failed for `1 + 2`"
        )
    }
}

class MyParameterizedTests {
    companion object {
        @JvmStatic
        fun calcInputs() = listOf(
            Arguments.of(1, 2, "+", 3),
            Arguments.of(2, 3, "-", -1)
        )
    }

    @ParameterizedTest
    @MethodSource("calculatorInputs")
    fun testCalculator(a: Int, b: Int, op: String, expected: Int) {
        assertEquals(expected, myCalculator(a, b, op), "Failed for `$a $op $b`")
    }
}
```

Annotations

`JUnit5` framework is the most popular way to test Java and Kotlin programs. There are a lot of annotations to customize tests:

- `@BeforeEach` -- methods annotated with this annotation are run before each test in the class
- `@AfterEach` -- methods annotated with this annotation are run after each test in the class
- `@BeforeAll` -- methods annotated with this annotation are run before all tests in the class
- `@AfterAll` -- methods annotated with this annotation are run after all tests in the class

```kotlin
class MyTestsWithInitialization {
```

```
    lateinit var calculator: Calculator

    @BeforeEach
    fun setUp() { calculator = Calculator(); ... }

    @ParameterizedTest
    @CsvFileSource(files = ["testCalculatorInputs.csv"])
    fun `test calculator from CSV`(a: Int, b: Int, op: String, expected: Int) {
        assertEquals(expected, calculator.op(op)(a, b), "Failed for `$a $op $b`")
    }

    @Test
    fun `division by zero should cause an exception`() {
        val exception: Exception = assertThrows(ArithmeticException::class.java) {
            calculator.op("/")(1, 0)
        }
        assertEquals("/ by zero", exception.message)
    }
}
```

23-02-28

Profiling