

Сахар Котлина

Делаем жизнь слаще с Котлин фишками

Содержание

1. Поздняя инициализация переменной
2. Инициализация по требованию
3. with, let, also, apply, run
4. use, lambda

1. Поздняя инициализация переменной

В джава у нас с переменными было все просто – она или хранит null или что-то. Ее можно менять по ходу дела или нет. Все (хотя не все конечно, кроме сильных ссылок есть еще мягкие и фантомные ссылки, но это слишком редко используется и всегда порождает больше вопросов чем ответов, можете изучить сами java soft reference, phantom reference). До сих пор в джава мы использовали лишь strong reference.

Теперь же давайте вернемся к котлин. Мы говорим что можем хранить в ссылке объект и говорим что можно запретить хранение null. А значит что мы не сможем получить NullPointerException. Вообще да, но есть один нюанс. В котлин есть такая классная штука как поздняя инициализация переменной. Предположим что вы объявили переменную, но не инициализируете ее в конструкторе и не инициализируете ее в init блоке и не там где инстанцируете(`val a = A()` - место инстанциации(объявления) и мгновенная инициализация)

```
object Main {  
    @JvmStatic  
    fun main(args: Array<String>) {  
        val a = A()  
        a.doAny()  
    }  
}  
  
class A {  
    private lateinit var b : B  
  
    fun doAny() {  
        b.doSome()  
    }  
}  
  
class B {  
    fun doSome() {  
        println("call b")  
    }  
}
```

Класс В простой класс с 1 методом. В классе А мы написали поле класса типа В но как видите в нем lateinit var – что это значит? Что мы обещаем инициализировать ее позже. Но как видите в методе класса А мы вызываем сразу метод у класса В и угадайте что будет когда мы вызовем метод у класса А – правильно, ошибка

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...  
Exception in thread "main" kotlin.UninitializedPropertyAccessException: lateinit property b has not been initialized  
    at lazy.A.doAny(Main.kt:20)  
    at lazy.Main.main(Main.kt:11)  
Process finished with exit code 1
```

Мы обещали инициализировать переменную но забыли об этом, ай-яй-яй. Предположим у нас нет такой фишки как lateinit. Как переписать код? Вариант первый – через нулабл поле

```
class A {  
    private var b : B? = null  
  
    fun doAny() {  
        b?.doSome()  
    }  
}
```

Если теперь запустить мейн то никаких ошибок не будет. Мы безопасно вызвали метод у объекта который может хранить (и хранит) нул. В чем минус такого подхода? Что мы всегда будем вызывать безопасно методы класса и не будем точно знать что в нем нул или нет. Еще вариант – инициализировать сразу.

```
class A {  
    private var b : B = B()  
  
    fun doAny() {  
        b.doSome()  
    }  
}
```

Но а что если в классе А кроме этого метода есть еще методы и не факт что нам понадобится это поле? Или например так

```
class A {  
    private var b : B = B()  
  
    fun doAny(flag: Boolean = false) {  
        if (flag)  
            b.doSome()  
    }  
}
```

Мы вызываем метод у класса В только тогда, когда удовлетворяются условия. А если они не удовлетворяются? Зачем нам порождать объект классас В? Притом что это занятие может быть дорогим. Предположим класс В имеет 5 полей и каждое из них само инициализируется

через другие классы и в итоге наши ресурсы будут зря потрачены (CPU) ибо в методе придет false и наш объект будет ненужным грузом. Как еще можно сделать? Например так

```
class A {  
    private var b: B? = null  
  
    fun doAny(flag: Boolean = false) {  
        if (flag) {  
            if (b == null)  
                b = B()  
            b!!.doSome()  
        }  
    }  
}
```

Если нам нужен объект, но он нул, то мы просто инициализируем его по-быстрому и вызывем метод через !! чтобы зря не проверять еще раз на нул. Что плохого в таком коде? Что у нас опять же переменная nullable и приходится писать некрасивый код с !!.

Ладно, что же можно сделать тогда? Сдержать обещание! Зачем же мы писали lateinit var

```
class A {  
    private lateinit var b: B  
  
    fun doAny(flag: Boolean = false) {  
        if (flag) {  
            b = B()  
            b.doSome()  
        }  
    }  
}
```

Мы обещали инициализировать объект b и мы сдержали обещание – как только нам понадобился объект мы его по-быстрому инициализировали и вызываем метод спокойно без ?. и без !!. красота же ну.

Но в чем проблема такого подхода? А что если мы 2 раза вызовем метод? Тогда 2 раза будет создан объект? Проверим на деле. Залогим конструктор класса B и вызовем метод класса A 2 раза.

```
@JvmStatic  
fun main(args: Array<String>) {  
    val a = A()  
    a.doAny(flag: true)  
    a.doAny(flag: true)  
}
```

B

lazy.Main x

/usr/lib/jvm/java-1.8.0-openjdk-amd64

b instantiatedcall b
b instantiatedcall b
|
Process finished with exit code 0

Да, так и есть. 2 раза вызвался конструктор и вся прелесть решения канула в лету. Так что кажется что самым лучшим решением это

```
class A {  
    private var b: B? = null  
  
    fun doAny(flag: Boolean = false) {  
        if (flag) {  
            if (b == null)  
                b = B()  
            b!!.doSome()  
        }  
    }  
}
```

Хотя если вы уверены что ваш метод doAny в классе A будет вызываться лишь единожды, то lateinit var ваше решение.

Еще раз – нет плохих или хороших решений. Есть более подходящее под задачу.

Но подождите, вы же не хотите сказать, что в котлин не придумали более изящного решения чтобы объект не создавался более 1 раза и не надо было писать нулабл некрасивый код?

2. Инициализация по требованию

Ну естественно, в котлин есть супер фишка которая решит ваши проблемы – вы не получите UninitializedPropertyAccessException и вам не нужно писать нулабл код.

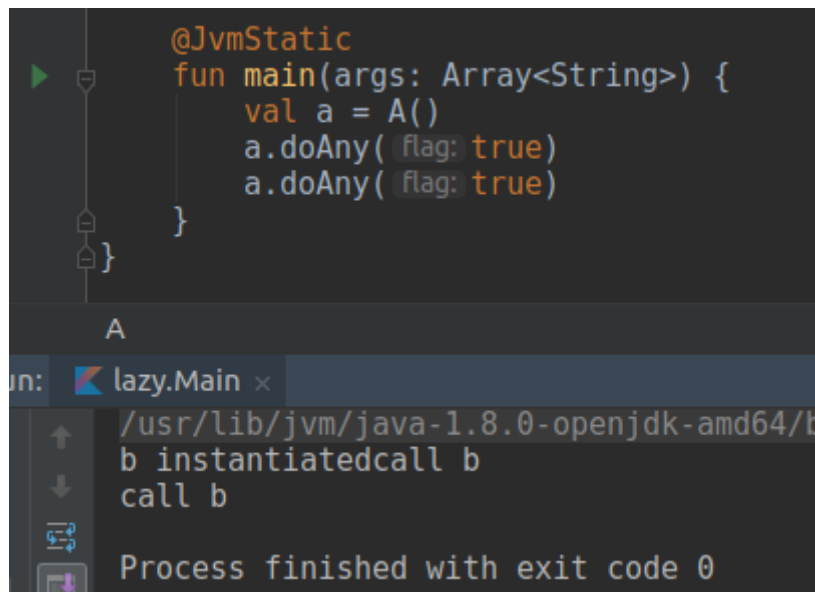
Если в котлин есть поздняя инициализация, где мы обещаем инициализировать позже нашу переменную (кстати нередко используется в андроид), то так же должна быть такая возможность как инициализация по требованию. Т.е. пока никто не нуждается в переменной ее не будем инициализировать. Давайте посмотрим на такой код.

Private val b by lazy { B() } что это такое – финальная переменная которая инициализируется лениво. Т.е. тогда когда нужно, не раньше и не позже.

```
class A {  
    private val b by lazy {  
        B()  
    }  
  
    fun doAny(flag: Boolean = false) {  
        if (flag) {  
            b.doSome()  
        }  
    }  
}
```

Мы пишем в объявлении переменной каким способом инициализировать переменную, но

оставляем на потом вызов этого (метода?) куска кода. Теперь давайте запустим мейн и посмотрим на логи. Ведь ранее мы создавали 2 объекта класса B, теперь не будет такого?



```
@JvmStatic
fun main(args: Array<String>) {
    val a = A()
    a.doAny( flag: true)
    a.doAny( flag: true)
}
```

Process finished with exit code 0

Нет, все прекрасно! Однозначно лучшая фишка котлина. Вопрос, мы бы могли написать такой же код в Java? Это было бы похоже на проверку на нол все равно. Можно в джава еще через синглтон, но надо его руками занулять. Ведь однажды созданный объект в синглтоне не будет сам по себе удален из памяти, нужно его руками занулять. Именно поэтому и мы не рекомендуем к использованию синглтон в джава коде если ваш синглтон не должен жить все время работы программы.

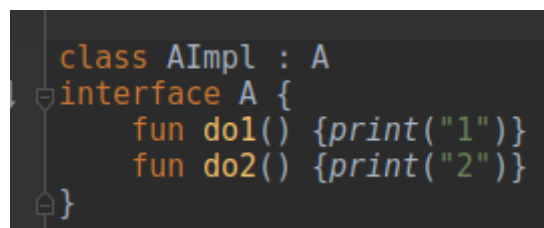
Ну и конечно же не все так просто с `by lazy` – вам не нужно использовать его для доступа к полям класса которые не должны жить дольше, ведь тогда вы создадите проблему. Это чаще может возникнуть в андроид и потому мы поговорим об этом в разделе андроид. А пока просто будьте внимательны при использовании этой фишки. Не делайте сложных вещей без понимания.

И напоследок давайте рассмотрим еще несколько классных фишек котлина

3. with, let, also, apply, run

Есть несколько классных операторов в Котлин, которые просто упрощают написание кода. Я приведу примеры на джава и на котлин чтобы было понятно где преимущества котлина.

1. With



```
class AImpl : A
interface A {
    fun do1() {print("1")}
    fun do2() {print("2")}
}
```

Предположим в вашем классе много методов и вам надо вызвать их подряд.

```
public static void main(String[] args) {
    A a = new AImpl();
    a.do1();
    a.do2();
}
```

А теперь котлин код. Мы меняем контекст и вызываем методы класса без указания каждый раз что они принадлежат ему. Так просто, да. В джава это просто невозможно.

```
with(AImpl()) { this: AImpl
    do1()
    do2()
}
```

2. **let** : то же самое что и with но можно передать ссылку на объект

```
class B {
    fun doSome(a: A) {
        print("some")
        a.do1()
    }
}
```

Сначала джава код

```
public static void main(String[] args) {
    A a = new AImpl();
    a.do2();
    new B().doSome(a);
}
```

Как видите нам опять нужна переменная чтобы сначала у нее вызвать метод и потом передать во внутрь метода класса B. А теперь посмотрите как просто это делать в котлин.

```
@JvmStatic
fun main(args: Array<String>) {
    AImpl().let { it: AImpl
        it.do2()
        B().doSome(it)
    }
}
```

Легко понять когда использовать let – Если вам нужно внутри вызвать метод этого класса и передать его же кому-то внутри другому. Как видите it это AImpl. Еще одна классная штука в котлин, можно дать другое название если вас смущает слово it

```
fun main(args: Array<String>) {
    AImpl().let { a ->
        a.do2()
        B().doSome(a)
    }
}
```

Вам надо было бы инстанциировать переменную a, но вместо этого вы юзаете let.

3. **also** похоже на let

В основном используем в таких ситуациях.

```
private static A getA() {  
    A a = new AImpl();  
    a.do2();  
    new B().doSome(a);  
    return a;  
}
```

Разница с let в том, что let не возвращает объект.

```
fun getA() : A = AImpl().also { it: AImpl  
    it.do2()  
    B().doSome(it)  
}
```

т.е. здесь вы бы не могли использовать let потому что метод должен вернуть объект A.

4. **apply** похож на **also** за исключением того что не надо передавать себя как it

```
fun getList() = ArrayList<String>().apply { this: ArrayList<String>  
    add("1")  
    add("2")  
}
```

т.е. при выборе apply или let надо смотреть на то, с чем мы работаем после использования.

То же самое в джава пишется примерно вот так

```
private static List<String> getList() {  
    List<String> list = new ArrayList<>();  
    list.add("1");  
    list.add("2");  
    return list;  
}
```

5. **run** похож на **with**

Лишь за исключением того что вы можете использовать run если не уверены что объект не нул

```
main(args: Array<String>): Unit {  
    val a: A? = null  
    with(a) { this: A?  
        a?.do1()  
        a?.do2() ^with  
    }  
}
```

Как видите толку от такого кода нет. А теперь посмотрите на `run`

```
val a: A? = null
a?.run { this: A
    do1()
    do2()
}
```

Совсем другое дело. Точно так же можно и `let` юзать у нулабл переменной и внутри уже точно не будет нул. Код на джаве думаю понимаете как будет

```
final A a = null;
if (a != null) {
    a.do1();
    a.do2();
}
```

Вложенность и там и там, но в джава мы не можем вызвать методы один за другим.

4. use, lambda

use как замена `try with resources` Java 8

Если помните мы должны были закрывать стрим данных после завершения.

Можете открыть лекцию 31 и посмотреть на тот код на джава и теперь вот так на котлин. Метод `copyTo` как видите из джавадока (java doc документация джава кода – все что написано над методом или классом в стиле `/** */` где описан класс или метод с его аргументами и возвращаемым типом, в отличие от блока комментария `/*...*/` можно просматривать снаружи класса через `quick doc`) должен закрывать оба стрима снаружи. Т.е. у нас стрим на выход и на вход который надо закрыть там, где мы вызываем этот метод. Ведь он по сути не делает это за нас. Потому вот такой двойной финт ушами.

```
@JvmStatic
fun main(args: Array<String>) {
    val url = "https://file-examples-com.github.io/uploads/2017/04/file_example_MP4_1920_18MG.mp4"

    BufferedInputStream(URL(url).openStream()).use { it: BufferedInputStream
        File("someVideo.mp4").copyInputStreamToFile(it)
    }
}

fun File.copyInputStreamToFile(inputStream: InputStream) {
    this.outputStream().use { fileOut ->
        inputStream.copyTo(fileOut)
    }
}
```

В нашей экстеншн функции мы принимает стрим и закрываем лишь внутренний, который относится к файлу. А значит надо закрыть внешний, который приняли аргументом. Потому вот так вот. Значит мы закроем стрим файла сперва и потом уже стрим из ссылки.


```

/**
 * Copies this stream to the given output stream, returning the number of bytes copied
 *
 * Note It is the caller's responsibility to close both of these resources.
 */
public fun InputStream.copyTo(out: OutputStream, bufferSize: Int = DEFAULT_BUFFER_SIZE): Long {
    var bytesCopied: Long = 0
    val buffer = ByteArray(bufferSize)
    var bytes = read(buffer)
    while (bytes >= 0) {
        out.write(buffer, 0, bytes)
        bytesCopied += bytes
        bytes = read(buffer)
    }
    return bytesCopied
}

```

Но давайте напишем иначе немного, применим ООП подход

```

class Streams(private val inStream: InputStream,
              private val outputStream: OutputStream,
              private val streamHandler: (InputStream, OutputStream) -> Unit) {

    init {
        inStream.use { it: InputStream
            outputStream.use { it: OutputStream
                streamHandler.invoke(inStream, outputStream)
            }
        }
    }
}

```

Напишем класс который принимает 2 стрима данных и напишем лямбду для того, чтобы снаружи можно было копировать данные. В инициализации вызовем код лямбды, который обернули в use первого и второго стрима. Т.е. когда мы создадим объект этого класса, то он сделает всю работу на старте и закроет оба стрима данных. На самом деле не очень правильно делать что-то такое при инициализации, по-хорошему нужен метод fun start().

Теперь посмотрите на вызов в мейне

Передаем оба стрима и пишем логику копирования как нам угодно.

```

@JvmStatic
fun main(args: Array<String>) {
    val url = "https://file-examples-com.github.io/uploads/2017/04/file_example_MP4_1920_18MG.mp4"

    Streams(BufferedInputStream(URL(url).openStream()),
            File("someVideo.mp4").outputStream()) { inputStream, outputStream ->
        inputStream.copyTo(outputStream)
    }
}

```

Но да, лучше написать нормальный метод

```

class Streams(private val inStream: InputStream,
              private val outStream: OutputStream,
              private val streamHandler: (InputStream, OutputStream) -> Unit) {

    fun start() = inStream.use { it: InputStream
        outStream.use { it: OutputStream
            streamHandler.invoke(inStream, outStream)
        }
    }
}

```

И вызвать когда нужно

```

@JvmStatic
fun main(args: Array<String>) {
    val url = "https://file-examples-com.github.io/uploads/2017/04/file_example_MP4_1920_18MG.mp4"

    Streams(BufferedInputStream(URL(url).openStream()),
            File("someVideo.mp4").outputStream()) { inputStream, outputStream ->
        inputStream.copyTo(outputStream)
    }.start()
}

```

И да, еще одна вещь – если наша лямбда последняя в списке аргументов конструктора, то можно его написать вне скобок. Обратите внимание, складывается ощущение что класс принял 2 аргумента – входной и выходной стримы и потом сделал что-то еще. Но нет, пока вы не вызовете метод start ничего не произойдет и так правильно.

Наш класс Streams будет работать для любых 2 стримов на вход и выход и сам закроет за собой оба стрима. Если вы хотите именно класс для скачивания данных, то вам надо принять в конструкторе ссылку на файл и имя файла как аргументы. Все остальное написать внутри метода start.

Еще раз заострим внимание на лямбде – если мы написали интерфейс, то код не будет красивым, нужен объект который наследует интерфейс и получаем анонимный класс. А для красоты надо писать лямбду полноценную – тип `streamHandler: (T, R) → Unit` т.е. лямбда юзает 2 переменные и не возвращает ничего – Unit в kotlin это void в java. Если ваша лямбда должна вернуть например Boolean, то будет `(T, R) → Boolean` вот и все.