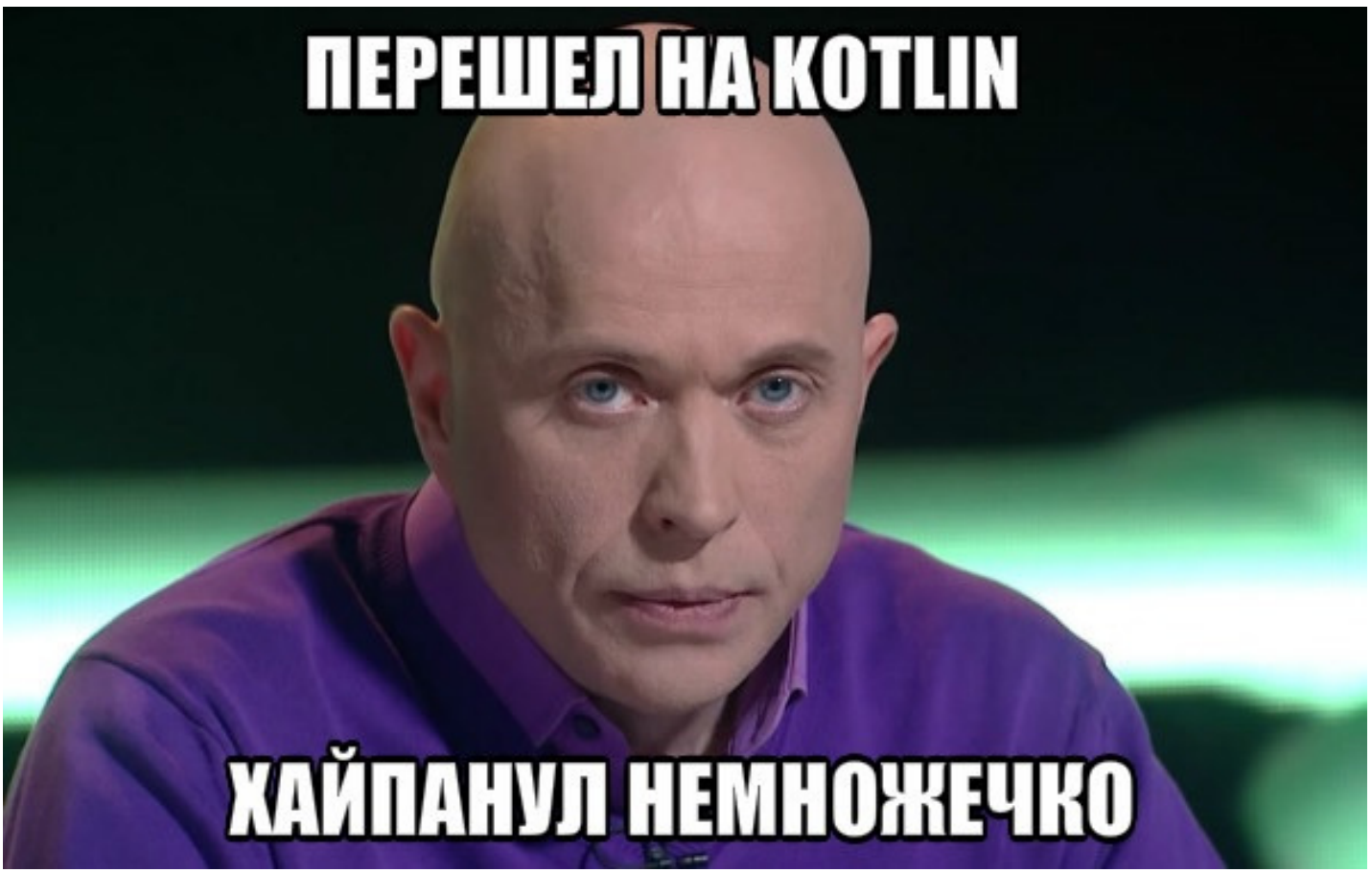
РАЗРАБОТКА ПОД ANDROID*, ПРОГРАММИРОВАНИЕ*, KOTLIN*, JAVA*, БЛОГ КОМПАНИИ MAIL.RU GROUP

Почему следует полностью переходить на Kotlin

ПЕРЕВОД

AloneCoder 24 мая 2017 в 19:01 68,9k

Оригинал: Magnus Vinther



Хочу рассказать вам о новом языке программирования, который называется Kotlin, и объяснить, почему вам стоит использовать его в своём следующем проекте. Раньше я предпочитал Java, но в последний год пишу на Kotlin везде, где только можно. И в данный момент я не представляю себе ситуации, в которой лучше было бы выбрать Java.

Kotlin разработан в JetBrains, и участие тех же людей в создании наборов IDE, таких как **IntelliJ** и **ReSharper**, хорошо заметно по самому языку. Он **прагматичен** и **краток**, благодаря чему написание кода превращается в приятный и эффективный процесс.

Хотя Kotlin компилируется в JavaScript и скоро будет компилироваться в машинный код, я сконцентрируюсь на его первичной среде — **JVM**.

Итак, несколько причин, почему вам следует полностью переходить на Kotlin (порядок случаен):

0# Совместимость с Java

Kotlin на **100 % совместим с Java**. Вы можете в буквальном смысле продолжать работать над своим старым Java-проектом, но уже используя Kotlin. Все ваши любимые **Java-фреймворки также будут доступны**, и, в каком бы фреймворке вы ни писали, Kotlin будет легко принят упрямым любителем Java.

1# Знакомый синтаксис

Kotlin — не какой-то там странный язык, рождённый в академических кругах. Его синтаксис знаком любому программисту, воспитанному на парадигме ООП, и с самого начала может быть более-менее понятным. Конечно, есть *некоторые* отличия от Java вроде переработанного конструктора

или объявлений переменных `val var`. В этом коде отражена большая часть базового синтаксиса:

```
class Foo {

    val b: String = "b"     // val means unmodifiable
    var i: Int = 0          // var means modifiable

    fun hello() {
        val str = "Hello"
        print("$str World")
    }

    fun sum(x: Int, y: Int): Int {
        return x + y
    }

    fun maxOf(a: Float, b: Float) = if (a > b) a else b

}
```

2# Интерполяция строк

Это как бы более умная и читабельная версия `String.format()` из Java, встроенная в язык:

```
val x = 4
val y = 7
print("sum of $x and $y is ${x + y}")  // sum of 4 and 7 is 11
```

3# Выведение типа

Kotlin будет выводить ваши типы, если вы посчитаете, что это улучшит читабельность:

```
val a = "abc"                           // type inferred to String
val b = 4                               // type inferred to Int

val c: Double = 0.7                     // type declared explicitly
val d: List<String> = ArrayList()       // type declared explicitly
```

4# Умные приведения типов (Smart Casts)

Компилятор Kotlin отслеживает вашу логику и **по мере возможности автоматически выполняет приведение типов**, т. е. вам больше не нужны проверки `instanceof` после явных приведений:

```
if (obj is String) {
    print(obj.toUpperCase())     // obj is now known to be a String
}
```

5# Интуитивные равенства (Intuitive Equals)

Можно больше не вызывать явно `equals()`, потому что оператор `==` теперь проверяет структурное равенство:

```
val john1 = Person("John")
val john2 = Person("John")
john1 == john2    // true  (structural equality)
john1 === john2   // false (referential equality)
```

6# Аргументы по умолчанию

Больше не нужно определять несколько одинаковых методов с разными аргументами:

```
fun build(title: String, width: Int = 800, height: Int = 600) {
    Frame(title, width, height)
}
```

7# Именованные аргументы

В сочетании с аргументами по умолчанию именованные аргументы избавляют от необходимости использовать Строителей:

```
build("PacMan", 400, 300)                                // equivalent
build(title = "PacMan", width = 400, height = 300)  // equivalent
build(width = 400, height = 300, title = "PacMan")  // equivalent
```

8# Выражение When

Оператор ветвления заменён гораздо более читабельным и гибким в применении выражением *when*:

```
when (x) {
    1 -> print("x is 1")
    2 -> print("x is 2")
    3, 4 -> print("x is 3 or 4")
    in 5..10 -> print("x is 5, 6, 7, 8, 9, or 10")
    else -> print("x is out of range")
}
```

Оно работает и как выражение (expression), и как описание (statement), с аргументом или без него:

```
val res: Boolean = when {
    obj == null -> false
```

```
    obj is String -> true
    else -> throw IllegalStateException()
}
```

9# Свойства

Можно добавить публичным полям кастомное поведение set & get, т. е. перестать набивать код безумными геттерами и сеттерами.

```
class Frame {
    var width: Int = 800
    var height: Int = 600

    val pixels: Int
        get() = width * height
}
```

10# Data Class

Он наполнен POJO-объектами `toString()`, `equals()`, `hashCode()` и `copy()`, но, в отличие от Java, не занимает 100 строк кода:

```
data class Person(val name: String,
                  var email: String,
                  var age: Int)

val john = Person("John", "john@gmail.com", 112)
```

11# Перегрузка оператора (Operator Overloading)

Заранее определённый набор операторов, которые можно перегружать для улучшения читабельности:

```
data class Vec(val x: Float, val y: Float) {
    operator fun plus(v: Vec) = Vec(x + v.x, y + v.y)
}

val v = Vec(2f, 3f) + Vec(4f, 1f)
```

12# Деструктурирующие объявления (Destructuring Declarations)

Некоторые объекты могут быть деструктурированы, что бывает полезно, к примеру, для итерирования map:

```
for ((key, value) in map) {
    print("Key: $key")
    print("Value: $value")
}
```

13# Диапазоны (Ranges)

Для улучшения читабельности:

```
for (i in 1..100) { ... }
for (i in 0 until 100) { ... }
for (i in 2..10 step 2) { ... }
for (i in 10 downTo 1) { ... }
if (x in 1..10) { ... }
```

14# Функции-расширения (Extension Functions)

Помните свой первый раз, когда пришлось сортировать `List` в Java? Вам не удалось найти функцию `sort()`, и пришлось изучать

`Collections.sort()`. Позднее, когда нужно было в строковом значении заменить все буквы строчными, вам пришлось писать собственную вспомогательную функцию, потому что вы не знали о `StringUtils.capitalize()`.

Если бы существовал способ добавления новых функций в старые классы, тогда ваш IDE помог бы найти правильную функцию при завершении кода. Именно это можно делать в Kotlin:

```
fun String.format(): String {
    return this.replace(' ', '_')
}

val formatted = str.format()
```

Стандартная библиотека расширяет функциональность оригинальных Java-типов, что особенно полезно для `String`:

```
str.removeSuffix(".txt")
str.capitalize()
str.substringAfterLast("/")
str.replaceAfter(":", "classified")
```

15# Безопасность Null

Java следует называть *почти* статично типизированным языком. Внутри него переменная типа `String` не *гарантированно* ссылается на `String` — она может ссылаться на `null`. И хотя мы к этому привыкли, это снижает безопасность проверки на статичное типизирование, и в результате Java-разработчики вынуждены жить в постоянном страхе перед NPE.

В Kotlin эта проблема решена посредством разделения на типы, **допускающие и не допускающие значение null**. По умолчанию типы не допускают null, но их можно преобразовать в допускающие, если добавить `?`:

```
var a: String = "abc"
a = null                // compile error

var b: String? = "xyz"
b = null                // no problem
```

Kotlin заставляет вас бороться с NPE, когда вы обращаетесь к типу, допускающему null:

```
val x = b.length        // compile error: b might be null
```

Возможно, выглядит громоздко, но благодаря нескольким своим возможностям действительно полезно. У нас всё ещё есть умные приведения типов, когда типы, допускающие null, преобразуются в не допускающие:

```
if (b == null) return
val x = b.length        // no problem
```

Также можно использовать безопасный вызов `?.`, он возвращает значение null вместо бросания NPE:

```
val x = b?.length       // type of x is nullable Int
```

Можно объединять безопасные вызовы в цепочки, чтобы избегать вложенных проверок если-не-null, которые иногда мы пишем в других языках. А если нам по умолчанию нужно не null-значение, то воспользуемся elvis-оператором `?:`:

```
val name = ship?.captain?.name ?: "unknown"
```

Если всё это вам не подходит и вам совершенно точно нужны NPE, то скажите об этом явно:

```
val x = b?.length ?: throw NullPointerException()  // same as below
val x = b!!.length                                 // same as above
```

16# Улучшенные лямбды

Это хорошая система лямбд —идеальный баланс между читабельностью и лаконичностью благодаря нескольким толковым решениям. Синтаксис прост:

```
val sum = { x: Int, y: Int -> x + y }   // type: (Int, Int) -> Int
val res = sum(4,7)                      // res == 11
```

А вот и толковые решения:

1. Можно переместить или опустить круглые скобки в методе, если лямбда идёт последней либо является единственным аргументом метода.

2. Если вы решили не объявлять аргумент одноаргументной лямбды, то он будет неявно объявлен под именем `it`.

Комбинация этих факторов делает эквивалентными эти три строки:

```
numbers.filter({ x -> x.isPrime() })
numbers.filter { x -> x.isPrime() }
numbers.filter { it.isPrime() }
```

Это позволяет нам писать лаконичный функциональный код, вы только посмотрите на эту красоту:

```
persons
    .filter { it.age >= 18 }
    .sortedBy { it.name }
    .map { it.email }
    .forEach { print(it) }
```

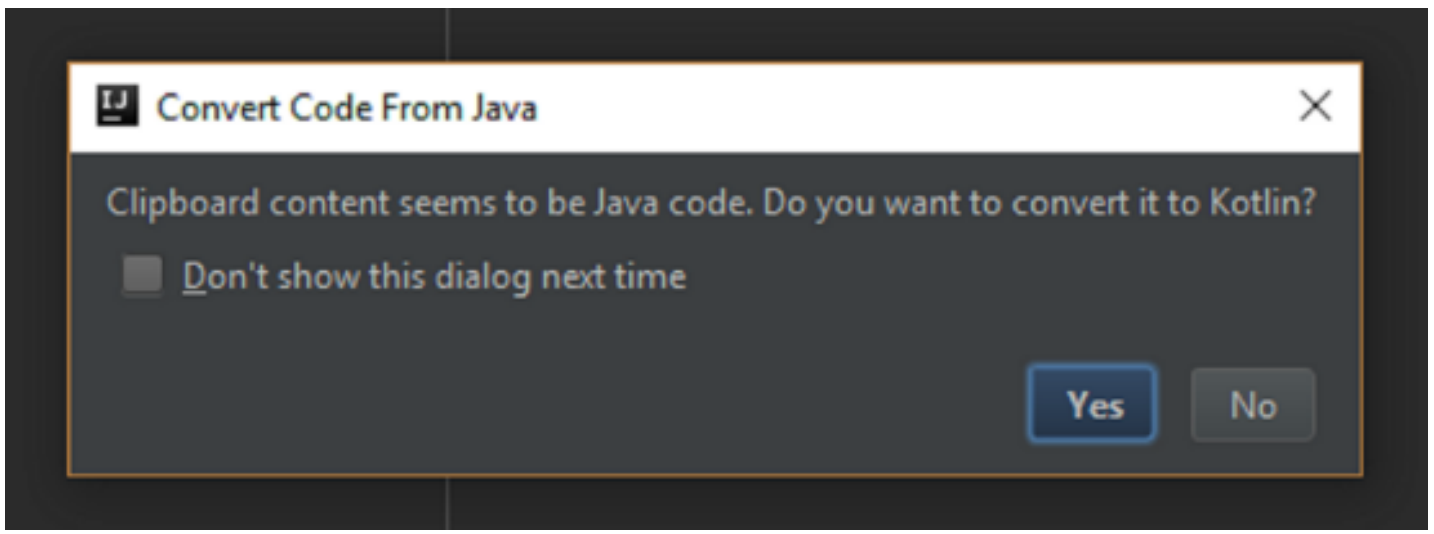
Система лямбд, объединённая с функциями-расширениями, делает Kotlin идеальным инструментом для создания DSL. Anko — пример DSL, предназначенного для расширения возможностей Android-разработки:

```
verticalLayout {
    padding = dip(30)
    editText {
        hint = “Name”
        textSize = 24f
    }
    editText {
        hint = “Password”
        textSize = 24f
    }
    button(“Login”) {
        textSize = 26f
```

```
    }
}
```

17# Поддержка IDE

У вас есть целый ряд вариантов, как можно начать работать с Kotlin, но я крайне рекомендую использовать **IntelliJ, идущий в комплекте поставки Kotlin** — его возможности демонстрируют преимущество ситуации, когда одни и те же люди разрабатывают как язык, так и IDE.
Небольшой пример: это сообщение всплыло, когда я впервые попытался скопипастить Java-код со Stack Overflow:



IntelliJ заметит, что вы вставляете Java-код в файл Kotlin

На этом всё, спасибо за чтение! Если мне пока не удалось убедить вас насчёт Kotlin, то рекомендую почитать дополнительные материалы:

- Kotlin on Android. Now official

- Why Kotlin is my next programming language
- Scala vs Kotlin
- Swift is like Kotlin
- The Road to Gradle Script Kotlin 1.0
- Introducing Kotlin support in Spring Framework 5.0
- 10 cool things about Kotlin
- Kotlin full stack application example
- Why I abandoned Java in favour of Kotlin
- I used Kotlin at Hackathon
- From Java to Kotlin
- Kotlin Idioms

Похожие публикации

21 совет по эффективному использованию Composer

7

ПЕРЕВОД

AloneCoder • 12 января в 15:49

Линейная регрессия с помощью Go

22

ПЕРЕВОД

AloneCoder • 25 декабря 2017 в 19:09

Рассмотрим Kotlin повнимательнее

24

ПЕРЕВОД

AloneCoder • 23 июня 2017 в 12:44

Популярное за сутки

Яндекс открывает Алису для всех разработчиков. Платформа Яндекс.Диалоги (бета)

69

BarakAdama • вчера в 10:52

Почему следует игнорировать истории основателей успешных стартапов

20

ПЕРЕВОД

m1rko • вчера в 10:44

Как получить телефон (почти) любой красотки в Москве, или интересная особенность MT_FREE

24

ИЗ ПЕСОЧНИЦЫ

cab404 • вчера в 20:27

Java и Project Reactor

10

zealot_and_frenzy • вчера в 10:56

Пользовательские агрегатные и оконные функции в PostgreSQL и Oracle 6

erogov • вчера в 12:46

Лучшее на Geektimes

Как фермеры Дикого Запада организовали телефонную сеть на колючей проволоке 31

NAGru • вчера в 10:10

Энтузиаст сделал новую материнскую плату для ThinkPad X200s 49

alizar • вчера в 15:32

Кто-то посылает секс-игрушки с Amazon незнакомцам. Amazon не знает, как их остановить 85

Pochtoycom • вчера в 13:06

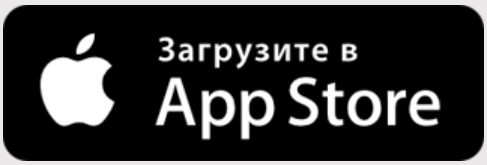
Илон Маск продолжает убеждать в необходимости создания колонии людей на Марсе 140

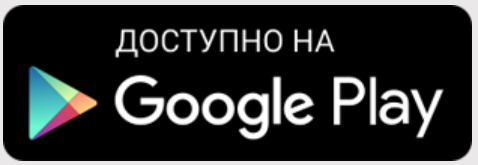
marks • вчера в 14:19

Дела шпионские (часть 1) 16

TashaFridrih • вчера в 13:16

Мобильное приложение





Полная версия