Kotlin for Android

План

- Сравнение java и котлин
- Что в котлине нового по сравнении с джавой
- Взаимодействие джавы и котлина

Kotlin

- Императивный, объектно ориентированный, функциональный
- Статическая типизация
- Полная совместимость с Java (interop)
- Задумывался
  - Лаконичнее чем Java
  - Проще чем Scala
- Разрабатывается JetBrains и сообществом

https://kotlinlang.org/

Kotlin на Android

- Официальная поддержка Android
- Support библиотека от Google с расширениями
- Быстро набирает популярность в Android тусовке
- Компактный и быстрый runtime
  - Размер Stdlib 964KB (версии 1.2.70).
  - Компилируется в байт-код, аналогично Java

https://kotlinlang.org/docs/tutorials/kotlin-android.html

Зачем нужен Kotlin на Android?

- Ограничения Java
  - Нерасширяемые классы из SDK aka ад из Util классов
  - Скудная Collection API
  - Проблемы с null
- Android поддерживается с Java 6 (Java 8 с Android 7.0)
  - Нет Java Date API
  - Нет Stream API
  - Нет нормальных лямбд
  - Try-with-resource (только с Android 4.4+)
  - Да и вообще Android SDK далеко не идеальный

Введение в синтаксис

- Определение переменных
- Функции
- Условные выражения
- Nullable
- Приведение типов
- Циклы, итерирование, проверка на вхождение
- Анонимные объекты и вспомогательные объекты

Определение переменных

Неизменяемая (только для чтения) переменная:

Kotlin	Java

`val a: Int = 1  // immediate assignment`
`val b = 2  // `Int` type is inferred`
`val c: Int`
`c = 3  // deferred assignment`
	`final Integer a = 1;`
	`final b = 2;  // compilation error`
	`final Integer c;`
	`c = 3;`

Определение переменных

Kotlin	Java

• Изменяемая переменная:
 	
`var x = 5 // `Int` type is inferred`
 	
`x += 1`
	• Изменяемая переменная:
	
	◦ `Integer x = 5;`
	
	◦ `x = 5;`

Объявление функции

Функция принимает два аргумента Int и возвращает Int:

Kotlin	Java

`fun sum(a: Int, b:Int): Int {`
 	
`    return a + b`
 	
`}`
	`public final int sum(int a, int b) {`
	
	`    return a + b;`
	
	`}`

Объявление функции (Kotlin feature)

Функция с выражением в качестве тела и автоматически определенным типом возвращаемого значения:

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

Объявление функции (Kotlin feature)

Функция, не возвращающая никакого значения (void в Java):

Kotlin	Java

`fun printSum(a: Int, b: Int) {`
`  print(a + b)`
`}`
	`public final void printSum( Int a, Int b) {`
	`  System.out.println(a + b)`
	`}`

Использование строковых шаблонов

Kotlin	Java

`fun main(args: Array<String>) {`
 	
`    if (args.size == 0) return`
 	
`    print("Первый аргумент: ${args[0]}")`
 	
`}`
	`public final void main(String[] args) {`
	
	`    if (args.length != 0) {`
	
	`        System.out.print("Первый аргумент: " + args[0]);`
	
	`    }`
	
	`}`

Использование условных выражений(Java way)

Kotlin	Java

`fun max(a: Int, b: Int): Int`
`{`
`  if (a > b)`
`  return a`
`  else`
`  return b`
`}`
	`public int max(Integer a,`
	`Integer b) {`
	`  if (a > b)`
	`  return a;`
	`  else`
	`  return b;`
	`}`

Использование условных выражений(Kotlin way)

Kotlin	Java

`fun max(a: Int, b: Int) =`
 	
`  if (a > b) a else b`
	`public int max(Integer a,`
	`Integer b) {`
	`  return (a > b) ? a : b;`

Nullable-значения и проверка на null

Ссылка должна быть явно объявлена как nullable (символ ?) когда она может принимать значение **null**.

```
fun parseInt(str: String): Int? {
    // ...
}
```

Возвращает **null** если **str** не содержит числа

Если из Java метод явно объявлен как nullable (@Nullable)

```
@Nullable
public Integer parseInt(String str) {
    // ...
}

val x: Int? = parseInt(null)
```

Nullable-значения и проверка на null

```
fun main(args: Array<String>) {
  if (args.size < 2) {
    print("Ожидается два целых числа")
    return
  }
  val x = parseInt(args[0])
  val y = parseInt(args[1])
  // Использование `x * y` приведет к ошибке, потому что они могут
содержать null
  if (x != null && y != null) {
    // x и y автоматически приведены к не-nullable после проверки на null
    print(x * y)
  }
}
```

Проверка типа и автоматическое приведение типов

```
fun getStringLength(obj: Any): Int? {
  if (obj is String) {
    // в этом блоке `obj` автоматически преобразован в `String`
    return obj.length
  }
  // `obj` имеет тип `Any` вне блока проверки типа
  return null
}
```

Проверка типа и автоматическое приведение типов

```
fun getStringLength(obj: Any): Int? {
    return if (obj is String && obj.length > 0) {
        return obj.length
    } else null
}
```

Использование цикла for

Kotlin	Java

`fun main(args: Array<String>){`
`  for (arg in args)`
`  print(arg)`
 	
`for (i in args.indices)`
`  print(args[i])`
`}`
	`public final void main(@NotNull String[] args) {`
	
	`  for(int i = 0; i < args.length; ++i) {`
	`  String arg = args[i];`
	`  System.out.print(arg);`
	`  }`
	
	`  int j = 0;`
	
	`  for(int k = args.length; j < k; ++j) {`
	`  String var6 = args[j];`
	`  System.out.print(var6);`
	`  }`

Использование выражения when

Kotlin

```
fun cases(obj: Any) {
  when (obj) {
    1          -> print("One")
    "Hello"    ->
print("Greeting")
    is Long    -> print("Long")
    !is String -> print("Not a
string")
    else       ->
print("Unknown")
  }
}
```

Java

```
public void cases(Object obj) {
    if (obj instance of Integer &&
(Integer)obj == 1) System.out.println("One")
    else if (obj instance of String &&
(String)obj.equals("Hello")) {

System.out.println("Greeting")

}   else if (obj instance of Long)
System.out.println("Long")

    else if (!(obj instance of String))
System.out.println("Not a string")

    else System.out.println("Unknown")
}
```

Использование интервалов

Проверка на вхождение числа в интервал с помощью оператора in:

Kotlin	Java

`if (x in 1..y-1)`
`  print("OK")`
	`if (x>= 1 && x<= y - 1) {`
	`  String var7 = "OK";`
	`  System.out.print(var7);`
	`}`

Использование интервалов

Проверка значения на выход за пределы интервала:

Kotlin	Java

`if (x !in 0..array.lastIndex)`
`  print("Out")`
	`if (array[array.length()-1] < x) {`
	`  String var4 = "Out";`
	`  System.out.print("Out");`
	`}`

Использование интервалов

Перебор значений в заданном интервале:

```
for (x in 1..5)
  print(x)
```

Или по арифметической прогрессии:

```
for (x in 1..10 step 2) {
  print(x)
}
for (x in 9 downTo 0 step 3) {
  print(x)
}
```

Использование коллекций

Итерация по коллекции:

Kotlin	Java

`for (name in names)`
`  println(name)`
	`Iterator iterator = names.entrySet().iterator();`
	`while(iterator.hasNext()) {`
	`Map.Entry name = (Map.Entry)iterator.next();`
	`System.out.println(name);`
	`}`

Использование коллекций

Итерация по коллекции:

Проверка, содержит ли коллекция данный объект, с помощью оператора in:

```
val items = setOf("apple", "banana", "kiwi")
when {
  "orange" in items -> println("juicy")
  "apple" in items -> println("apple is fine too")
}
```

Использование коллекций

Использование лямбда-выражения для фильтрации и модификации коллекции:

```
names.filter { it.startsWith("A") }
        .sortedBy { it }
        .map { it.toUpperCase() }
        .forEach { print(it) }
```

Создание экземпляров классов

```
//не требуется ключевое слово 'new'
val rectangle = Rectangle(5.0, 2.0)
val triangle = Triangle(3.0, 4.0, 5.0)
```

Анонимные объекты

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // ...
    }

    override fun mouseEntered(e: MouseEvent) {
        // ...
    }
})
```

Анонимные объекты

Если у супертипа есть конструктор, то в него должны быть переданы соответсвующие параметры. Множество супертипов может быть указано после двоеточия в виде списка, заполненного через запятую:

```
open class A(x: Int) {
    public open val y: Int = x
}

interface B {...}


val ab: A = object : A(1), B {
    override val y = 15
}
```

Анонимные объекты

Если всё-таки нам нужен просто объект без всяких там родительских классов, то можем указать:

```
val adHoc = object {
    var x: Int = 0
    var y: Int = 0
}
print(adHoc.x + adHoc.y)
```

Вспомогательные объекты

Объявление объекта внутри класса может быть отмечено ключевым словом **companion**:

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

Для вызова членов такого **companion** объекта используется имя класса:

```
val instance = MyClass.create()
```

Вспомогательные объекты

Такие члены вспомогательных объектов выглядят, как статические члены в других языках программирования. На самом же деле, они являются членами реальных объектов и могут реализовывать, к примеру, интерфейсы:

```
interface Factory<T> {

   fun create(): T

}
class MyClass {
    companion object : Factory<MyClass> {
        override fun create(): MyClass = MyClass()
    }
}
```

Вспомогательные объекты(Вызов из Java)

Если ничего не указать то вызывать придётся так:

```
MyClass myClass =  MyClass.Companion.create();
```

А если над методом `create()` навесить аннотацию `@JvmStatic:`

```
MyClass myClass =  MyClass.create();
```

Вспомогательные объекты(Вызов из Java)

Free your mind

Идиомы котлина

- Names params
- Read-only список/словарь
- Ленивые свойства
- Функции-расширения
- data class
- Вызов оператора при равенстве null
- Выполнение при неравенстве null

Значения по умолчанию для параметров функций

```
fun foo(a: Int = 0, b: String = "") { ... }
```

Read-only список/словарь

```
val list = listOf("a", "b", "c")

val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

Функции-расширения

```
fun String.spaceToCamelCase() { ... }

"Convert this to
camelcase".spaceToCamelCase()
```

Вызов оператора при равенстве null

```
val data = ...
val email = data["email"] ?: throw
IllegalStateException("Email is missing!")
```

Выполнение при неравенстве null

```
val data = ...

data?.let {
    ... // execute this block if
not null
}
```

Метки операторов break и continue

Любое выражение в **Kotlin** может быть помечено меткой **label**. Метки имеют идентификатор в виде знака @. Например: метки abc@, fooBar@ являются корректными

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (...)
        break@loop
    }
}
```

Модификаторы доступа

Для членов, объявленых в классе:

- **private** означает видимость только внутри этого класса (включая его членов);
- **protected** --- то же самое, что и private + видимость в субклассах;
- **internal** --- любой клиент *внутри модуля*, который видит объявленный класс, видит и его internal члены;
- **public** --- любой клиент, который видит объявленный класс, видит его public члены.

Синтаксис лямбда-выражений

```
/** A function that takes 2 arguments. */
public interface Function2<in P1, in P2, out R> : Function<R> {
    /** Invokes the function with the specified arguments. */
    public operator fun invoke(p1: P1, p2: P2): R
}
```

Синтаксис лямбда-выражений

```
val sum = { x: Int, y: Int -> x + y }

val sum: (Int, Int) -> Int = { x, y -> x + y }

val useSum = sum(1,2)

ints.filter { it > 0 } //Эта константа имеет тип '(it: Int) -> Boolean'
```

Если параметр лямбды не используется, разрешено применять подчеркивание вместо его имени (с 1.1)

```
map.forEach { _, value -> println("$value!") }
```

Высокоуровневые функции и лямбды

- Лямбда-выражение всегда заключено в фигурные скобки;
- Его параметры (если они есть) объявлены до знака `->` (допустимо не указывать параметры);
- Тело выражения следует после знака `->`.

```
fun <T, R> List<T>.map(transform: (T) -> R): List<R> {
    val result = arrayListOf<R>()
    for (item in this)
        result.add(transform(item))
    return result
}
```

```
val doubled = ints.map { it -> it * 2 }
```

Замыкания

Лямбда-выражение или анонимная функция (так же, как и локальная функция или object expression) имеет доступ к своему замыканию, то есть к переменным, объявленным вне этого выражения или функции. В отличае от Java, переменные, захваченные в замыкании, могут быть изменены:

```
var sum = 0
ints.filter { it > 0 }.forEach {
    sum += it
}
print(sum)
```

Расширения (extensions)

Функции-расширения = в качестве приставки пишем *возвращаемый тип*, то есть тип, который мы расширяем.

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' даёт ссылку на лист
    this[index1] = this[index2]
    this[index2] = tmp
}

val a = mutableListOf(1, 2, 3)
a.swap(0, 2) // 'this' внутри 'swap()' будет содержать
значение 'a'
```

Расширения (extensions)

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.setType("text/plain");
intent.putExtra(Intent.EXTRA_EMAIL,
"emailaddress@emailaddress.com");
intent.putExtra(Intent.EXTRA_SUBJECT, "Subject");
intent.putExtra(Intent.EXTRA_TEXT, "I'm email body.");

startActivity(Intent.createChooser(intent, "Send Email"));
```

Расширения (extensions)

```
public inline fun <T> T.apply(block: T.() -> Unit): T { block(); return this}
```

```
val intent = Intent(Intent.ACTION_SEND).apply {
    type = "text/plain"
    putExtra(Intent.EXTRA_EMAIL, "emailaddress@emailaddress.com")
    putExtra(Intent.EXTRA_EMAIL, "emailaddress@emailaddress.com")
    putExtra(Intent.EXTRA_SUBJECT, "Subject")
    putExtra(Intent.EXTRA_TEXT, "I'm email body.")
}
startActivity(intent)
```

Расширения (extensions)

```
public inline fun <T, R> T.run(block: T.() -> R): R { return block(this) }
```

```
request = request.newBuilder().run {
            addHeader(HEADER_AUTH,authHeaderValue(session))
             addHeader(HEADER_COMPANY_ID, companyIdValue)
             build()
        }
```

Расширения (extensions)

```
public inline fun <T, R> T.let(block: (T) -> R): R { return block(this) }


val result = str.let {
    print(it) // Argument
    42 // Block return value
}
```

Расширения (extensions)

```
public inline fun <T, R> T.also(block: (T) -> R): R { block(this); return this }
```

```
class FruitBasket {
    private var weight = 0
    fun addFrom(appleTree: AppleTree) {
        val apple = appleTree.pick().also { apple ->
            this.weight += apple.weight
            add(apple)
        }
        ...
    }
    ...
    fun add(fruit: Fruit) = ...
}
```

Расширения (extensions)

Если в классе есть и член в виде обычной функции, и функция-расширение с тем же возвращаемым типом, таким же именем и применяется с такими же аргументами, то **обычная функция имеет более высокий приоритет**. К примеру:

```
class C {
    fun foo() { println("member") }
}

fun C.foo() { println("extension") }
```

Расширения (extensions)

Однако, для экстеншн-функций совершенно нормально перегружать функции-члены, которые имеют такое же имя, но другую сигнатуру:

```
class C {
    fun foo() { println("member") }
}

fun C.foo(i: Int) { println("extension") }
```

Расширения (extensions)

Свойства-расширения = расширение объявленное на переменной или константе

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

НО

```
val Foo.bar = 1 // ошибка: запрещено инициализировать
значения в свойствах-расширениях
```

Классы данных

```
data class User(val name: String, val age: Int)
```

Такой класс называется *классом данных*. Компилятор автоматически извлекает все члены данного класса из свойств, объявленных в первичном конструкторе:

- пара функций **equals()/hashCode()**,
- **toString()** в форме **"User(name=John, age=42)"**,
- функции **componentN()**, которые соответствуют свойствам, в зависимости от их порядка либо объявления,
- функция **copy()**

Делегированные свойства

```
class Example {
    var p: String by Delegate()
}
```

val/var <имя свойства>: <Тип> by <выражение>.

Выражение после *by* — *делегат*: обращения (get(), set()) к свойству будут обрабатываться этим выражением. Делегат не обязан реализовывать какой-то интерфейс, достаточно, чтобы у него были методы get() и set() с определённой сигнатурой:

Делегированные свойства

```
class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return "$thisRef, спасибо за делегирование мне '${property.name}'!"
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value:
String) {
        println("$value было присвоено значению '${property.name} в
$thisRef.'")
    }
}
```

Стандартные делегаты

Ленивые свойства (lazy properties)

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}

fun main(args: Array<String>) {
    println(lazyValue)
    println(lazyValue)
}
```

Псевдонимы типов

Примечание: Псевдонимы типов доступны в Kotlin начиная с версии 1.1

Псевдонимы типов полезны, когда вы хотите сократить длинные имена типов, содержащих обобщения.

```
typealias NodeSet = Set<Network.Node>

typealias FileTable<K> = MutableMap<K, MutableList<File>>

typealias MyHandler = (Int, String, Any) -> Unit

typealias Predicate<T> = (T) -> Boolean
```

Вызов кода Java из Kotlin

Java type	Kotlin type
byte	kotlin.Byte
short	kotlin.Short
int	kotlin.Int
long	kotlin.Long
char	kotlin.Char
float	kotlin.Float
double	kotlin.Double
boolean	kotlin.Boolean

Java type	Kotlin type
java.lang.Object	kotlin.Any!
java.lang.Cloneable	kotlin.Cloneable!
java.lang.Comparable	kotlin.Comparable!
java.lang.Enum	kotlin.Enum!
java.lang.Annotation	kotlin.Annotation!
java.lang.Deprecated	kotlin.Deprecated!
java.lang.CharSequence	kotlin.CharSequence!
java.lang.String	kotlin.String!
java.lang.Number	kotlin.Number!
java.lang.Throwable	kotlin.Throwable!

Вызов кода Java из Kotlin

Java type	Kotlin read-only type	Kotlin mutable type	Loaded platform type
`Iterator<T>`	`Iterator<T>`	`MutableIterator<T>`	`(Mutable)Iterator<T>!`
`Iterable<T>`	`Iterable<T>`	`MutableIterable<T>`	`(Mutable)Iterable<T>!`
`Collection<T>`	`Collection<T>`	`MutableCollection<T>`	`(Mutable)Collection<T>!`
`Set<T>`	`Set<T>`	`MutableSet<T>`	`(Mutable)Set<T>!`
`List<T>`	`List<T>`	`MutableList<T>`	`(Mutable)List<T>!`
`ListIterator<T>`	`ListIterator<T>`	`MutableListIterator<T>`	`(Mutable)ListIterator<T>!`
`Map<K, V>`	`Map<K, V>`	`MutableMap<K, V>`	`(Mutable)Map<K, V>!`
`Map.Entry<K, V>`	`Map.Entry<K, V>`	`MutableMap.MutableEntry<K,V>`	`(Mutable)Map.(Mutable)Entry<K, V>!`

Вызов кода Java из Kotlin

- Java's wildcards:
  - Foo<? extends Bar> становится Foo<out Bar!>!,
  - Foo<? super Bar> становится Foo<in Bar!>!;
- Java сырые типы конвертятся в звёздочки
  - List становится List<*>!, i.e. List<out Any?>!.

Вызов кода Kotlin из Java

```kotlin
// example.kt
package demo

class Foo

fun bar() { ... }
```

```java
// Java
new demo.Foo();
demo.ExampleKt.bar();
```

Вызов кода Kotlin из Java

Имена можно задавать самостоятельно используя аннотацию @JvmName

```kotlin
@file:JvmName("DemoUtils")

package demo

class Foo

fun bar() { ... }
```

```java
// Java
new demo.Foo();
demo.DemoUtils.bar();
```

Вызов кода Kotlin из Java

В том числе изменять проперти для вызова из Java

```
@get:JvmName("x")
@set:JvmName("changeX")
var x: Int = 23
```

Вызов кода Kotlin из Java

```kotlin
class Foo @JvmOverloads constructor(x: Int, y: Double = 0.0) {
    @JvmOverloads fun f(a: String, b: Int = 0, c: String = "abc") { ... }
}
```

```
// Constructors:
Foo(int x, double y)
Foo(int x)

// Methods
void f(String a, int b, String c) { }
void f(String a, int b) { }
void f(String a) { }
```

Ссылки

- Kotlin Basic Syntax
- Kotlin idioms
- Kotlin Classes and Inheritance
- Kotlin books
- Kotlin Libs
- Kotlin Koans (online)
- Kotlin Koans (github)
- Android Development with Kotlin — Jake Wharton