

Android на Kotlin

# Функции высшего порядка, лямбды и extension-функции

---



# На этом уроке

1. Изучим функции высшего порядка, лямбды, extension-функции и делегаты.
2. Узнаем о хороших практиках программирования на Kotlin.

## Оглавление

[Лямбда-выражения](#)

[Функции высшего порядка](#)

[Функции-расширения](#)

[Функции-расширения в стандартной библиотеке Kotlin](#)

[Разница между apply, with, let, also и run](#)

[Использование let](#)

[Использование apply](#)

[Использование also](#)

[Использование with](#)

[Использование run](#)

[Использование функций в цепочке вызовов](#)

[Делегирование](#)

[Хорошие практики при программировании на Kotlin](#)

[Практика](#)

[Факультатив: extension-функции для View](#)

[Отображать и прятать клавиатуру](#)

[Показать Snackbar](#)

[Управлять видимостью View](#)

[Получить Bitmap из View](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемые источники](#)

# Лямбда-выражения

Функции в Kotlin часто используются как самостоятельные объекты. Их можно хранить в переменных и передавать как параметры в другие функции:

```
fun main(args: Array<String>) {  
    print(greetingFun)  
}  
  
val greetingFun = fun(): String {  
    return "Hello"  
}  
  
fun print(block: () -> String) {  
    println(block())  
}
```

В примере выше мы сохранили в переменной `greetingFun` функцию, которая возвращает строку. Функция `print()` в качестве параметра принимает функциональный тип, то есть переменную в виде функции. Такой тип не принимает параметров и возвращает строку. Внутри функции `print` вызываем переданную функцию и выводим результат её выполнения. В функции `main` мы вызвали функцию `print`, передав ей в качестве параметра переменную `greetingFun`, где хранилась анонимная функция — с ключевым словом `fun`, но без названия.

Для упрощённой записи таких анонимных, то есть не имеющих собственного названия, функций служат лямбда-выражения. Функция `greetingFun` записывается так:

```
val greetingFun = { "Hello" }
```

Фигурными скобками мы обозначаем блок кода для выполнения, а Kotlin уже понимает, что это анонимная функция, возвращающая строку типа `fun(): String { return "Hello" }`.

Лямбда-выражения могут принимать параметры. Выражение `fun(a: Int, b: Int): Int { return a + b }` будет выглядеть так:

```
val sum = { a: Int, b: Int -> a + b }
```

Выражение всегда заключается в фигурные скобки. Его параметры отделяются от тела стрелкой. Если лямбда-выражение не принимает параметров, в фигурных скобках указывается только тело функции. Возвращаемое значение выражения — это результат последнего действия. Если требуется вернуть другое значение, можно использовать слово `return`, но с указанием метки, у которой в лямбда-выражении имя ближайшей функции:

```
repository.getNoteById(noteId).observeForever { t ->
    if (t == null) return@observeForever

    when (t) {
        is Success<*> ->
            viewStateLiveData.value = NoteViewState(note = t.data as? Note)
        is Error ->
            viewStateLiveData.value = NoteViewState(error = t.error)
    }
}
```

Если лямбда-выражение — единственный аргумент функции, круглые скобки можно опустить. А если оно не одно, но передаётся последним параметром, это лучше вынести за скобки:

```
snackbar.setAction(R.string.ok_bth_title) { snackbar.dismiss() }
```

Если лямбда принимает единственное значение, оно может не указываться в начале, а в теле выражения будет представлено именем `it`:

```
notesReference.document(id).get()
    .addOnSuccessListener {
        result.value = NoteResult.Success(it.toObject(Note::class.java))
    }.addOnFailureListener {
        result.value = NoteResult.Error(it)
    }
```

Здесь в `onSuccessListener` приходит значение типа `DocumentSnapshot`, а в `onFailureListener` — типа `Exception`. Каждое значение в своём лямбда-выражении представлено именем `it`.

`OnSuccessListener` и `OnFailureListener` — интерфейсы с одним методом, или SAM-интерфейсы. Их можно заменить лямбда-выражениями. Ещё пример:

```
someView.setOnClickListener { someAction() }
```

У лямбда-выражений есть доступ к внешним переменным. Если лямбда-выражение вызвано в функции, ему доступны переменные, объявленные в ней:

```
fun bind(note: Note) {
    title.text = note.title
    body.text = note.note
    /.../
    itemView.setOnClickListener { onItemClickListener(note) }
}
```

## Функции высшего порядка

Так называются функции, которые принимают другие функции в качестве параметров:

```
fun print(block: () -> Any) {  
    println(block())  
}
```

Функция `print` принимает параметр `block`, представляющий собой функциональный тип, вызывает выполнение переданной функции и выводит в консоль результат её работы.

Переменные тоже могут иметь функциональный тип:

```
private val onItemClickListener: (par: Parameter) -> Unit
```

Мы видим, что в качестве типа для переменной указано выражение `(par: Parameter) -> Unit`. Такие переменные передаются в функцию высшего порядка или вызываются напрямую как функция:

```
itemView.setOnClickListener { onItemClickListener(parameter) }
```

## Функции-расширения

Одна из полезнейших особенностей Котлина — возможность дополнять инструментарий классов, не расширяя их (не наследуясь) и не создавая классы, реализующие паттерн «Декоратор». Этим целям в Kotlin служат специальные функции-расширения. Они объявляются за пределами класса, но их можно вызывать как будто эти функции написаны в самом классе. Даже если этот класс закрыт для редактирования и находится в составе сторонней библиотеки. Такой подход не нов и реализован во многих языках.

К примеру, нам нужно отображать дату в нашем приложении в определенном формате. **Число-Месяц-Год Часы-Минуты: 9 января 21 12:34**. Дата в этом формате отображается на разных экранах нашего приложения и постоянно используется. Стандартным подходом в Джаве будет создание конкретного метода, который принимает дату в виде класса `Date`, преобразует ее в читаемый формат и возвращает в качестве `String` для отображения в текстовом поле. Сам метод `String format(Date date) {...}` будет находиться в некоем утилитарном статическом классе типа `DataUtils`.

В Котлине же мы можем написать функцию, расширяющую функционал класса `Date` (функция-расширение) и обращаться к этой функции через `Date`, как будто эта функция там была всегда. Чтобы добавить в класс `Date` функцию `format()`, напомним код:

```
const val DATE_TIME_FORMAT = "dd.MMM.yy HH:mm"

fun Date.format(): String =
    SimpleDateFormat(DATE_TIME_FORMAT, Locale.getDefault())
        .format(this)
```

Чтобы создать функцию-расширение, надо добавить перед именем функции наименование класса, который хотим расширить. Имя класса в этом случае называется типом-получателем. В теле функции можно напрямую обращаться к членам класса (через `this`), представляющего собой тип-получатель, как если бы эта функция объявилась внутри класса. Но так как функция объявлена снаружи класса, доступ внутри неё есть только к публичным членам класса.

Внутри этой функции можно обращаться к объекту, у которого вызвана эта функция, с использованием `this` — так же, как делали бы внутри класса.

Создав такую функцию, воспользуемся ею, будто она член класса:

```
val date = Date(1234)
dateTextView.text = date.format()
```

Мы создали некую дату, значит, мы можем вызвать у `Date` функцию-расширение `format()`.

Если в классе `Date` уже есть такая функция, то в месте вызова будет вызвана функция из класса. Но мы можем перегружать функции, объявленные внутри класса, изменяя сигнатуру.

Чтобы понять, как работают функции-расширения, достаточно взглянуть на Java-код, в который они компилируются. Для удобства удалим проверки, добавляемые компилятором Kotlin в сгенерированный код.

```

@NotNull
public static final String format(@NotNull Date $receiver) {
    String var10000 = (new SimpleDateFormat("dd.MMM.yy HH:mm",
        Locale.getDefault())) .format($receiver);
    return var10000;
}

```

Это обычная статическая функция, принимающая в качестве параметра объект-приёмник. Её вызов из кода на Java выглядел бы так:

```
dateTextView.setText(ExtensionsKt.format(date);
```

Она объявлена как функция верхнего уровня в файле Extensions.kt. Из Java-кода функции и переменные, объявленные в таких файлах, видны как члены класса с именем файла и окончанием Kt.

## Функции-расширения в стандартной библиотеке Kotlin

В стандартной библиотеке определено множество функций-расширений, которые делают работу с кодом на Kotlin удобнее. Применяются подходы функционального программирования. Рассмотрим наиболее часто используемые: `apply`, `with`, `let`, `also`, `run`. Посмотрим, как их использовать и когда, что между ними общего, какие различия. Они могут показаться одинаковыми в некоторых случаях, так как делают похожие вещи: принимают аргумент от ресивера (об этом ниже) и код, а затем применяют код на ресивере.

Посмотрим на простом примере функции `with`. Так выглядит код без функции:

```

class Person {
    var name: String? = null
    var age: Int? = null
}

val person: Person = getPerson()
print(person.name)
print(person.age)

```

А теперь с `with`:

```

val person: Person = getPerson()
with(person) {
    print(name)
    print(age)
}

```

Коротко и красиво! Теперь разберёмся, зачем используются остальные.

## Разница между apply, with, let, also и run

Возможности этих функций похожи, но есть разница в их сигнатурах и имплементации, которые и определяют границы их использования. Для начала сравним `with` и `also`. Эти функции в Котлине определяются так:

```
inline fun <T, R> with(receiver: T, block: T.() -> R): R {
    return receiver.block()
}
inline fun <T> T.also(block: (T) -> Unit): T {
    block(this)
    return this
}
```

Главная разница в том, что `with` принимает два аргумента: ресивер — объект, на который применяется эта функция, и функцию, возвращающая какой-то новый объект. А `also` принимает ресивер неявно в виде функции и возвращает результат выполнения блока кода в виде всё того же ресивера. Код будет отличаться:

```
val person: Person = getPerson()
with(person) {
    print(name)
    print(age)
}

val person: Person = getPerson().also {
    print(it.name)
    print(it.age)
}
```

В случае с `also` переменной `person` будет присвоено значение из метода `getPerson()`. Но перед этим `also` выведет в консоль имя и возраст. То есть через `also` можно произвести какие-то операции с `Person` заранее: вывести на экран. А `with` просто принимает уже созданный `person` в качестве аргумента и затем производит над ним какие-то изменения, не возвращая никакого результата.

Теперь посмотрим, как объявлены другие функции, и сравним их:

```
inline fun <T, R> with(receiver: T, block: T.() -> R): R {
    return receiver.block()
}
inline fun <T> T.also(block: (T) -> Unit): T {
    block(this)
    return this
}
inline fun <T> T.apply(block: T.() -> Unit): T {
    block()
}
```



```

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

```

Для лучшего усвоения можно воспользоваться шпаргалкой:

Kotlin Standard Scoping Functions			
input	receiver		parameter
binding in lambda			
receiver	apply	run	with
parameter	also	let	
output	same object	result of lambda	
Definitions			
fun <T, R> T.run(block: T.() -> R): R = block()			
fun <T, R> with(receiver: T, block: T.() -> R): R = receiver.block()			
fun <T> T.apply(block: T.() -> Unit): T { block(); return this }			
fun <T> T.also(block: (T) -> Unit): T { block(this); return this }			
fun <T, R> T.let(block: (T) -> R): R = block(this)			
Examples			
val r: R = T().run { this.foo(); this.toR() }			
val r: R = with(T()) { this.foo(); this.toR() }			
val t: T = T().apply { this.foo() }			
val t: T = T().also { it.foo() }			
val r: R = T().let { it.foo(); it.toR() }			

## Использование let

Лидер по частоте использования — функция `let`. Она определена в файле `Standard.kt` стандартной библиотеки Kotlin:

```

public inline fun <T, R> T.let(block: (T) -> R): R {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    return block(this)
}

```

Это функция-расширение, имеющая тип-приёмник T.

T — generic-параметр, он определится компилятором по типу, у которого вызвана эта функция. Функция принимает в качестве параметра функциональный тип, принимающий тип T и возвращающий тип R. Возвращает результат выполнения переданной функции, передав в неё объект, на котором вызвана функция. Вызов функции `contract` — это платформенная проверка, здесь его рассматривать не будем.

Обратим внимание на модификатор `inline`. Он сообщает компилятору, что код переданной в качестве параметра функции должен устроиться прямо в месте вызова функции `let` без создания статической функции. Поэтому использование этой функции обходится без дополнительных расходов во время исполнения программы.

Функция `let` обычно используется для проверки на `null`:

```
personId?.let {  
    loadPerson(it)  
}
```

Чтобы использовать `id` человека, вызываем оператор безопасного вызова «?». Если `personId != null`, будет вызвана функция `let` с лямбда-выражением в фигурных скобках, а `personId` — передана в это выражение в качестве единственного параметра. В лямбда-выражении значение `personId` представлено как `it`. Можно также задавать свое название для параметра `it`, чтобы сделать код более читабельным или где одна функция `let` вызывается внутри другой функции `let`:

```
personId?.let { id ->  
    loadPerson(id)  
}
```

Это идиоматичный для Kotlin способ работы с nullable-типами.

## Использование `apply`

Рассмотрим ещё одну часто используемую функцию:

```
public inline fun <T> T.apply(block: T.() -> Unit): T
```

Здесь не требуется приводить тело функции, её работу можно понять из сигнатуры. Функция имеет тип-приёмник T и принимает в качестве параметра функцию с тем же типом-приёмником. В теле функции вызывается переданная функция, а по завершении выполнения возвращается объект, на котором она была вызвана. То есть мы производим какие-то операции над объектом и возвращаем его. Распространённый сценарий использования этой функции — своеобразная замена паттерна

«Строитель/Builder». После вызова конструктора и перед возвратом готового объекта производится его конфигурация:

```
fun getStartIntent(context: Context, string: String?) =
    Intent(context, Activity::class.java).apply {
        putExtra(EXTRA_STRING, string)
    }

val peter = Person().apply {
    name = "Peter"
    age = 18
}
```

Здесь мы создаём новый экземпляр `Intent` и вызываем у него функцию `apply`. Внутри функции производим требуемую конфигурацию созданного `Intent` и только потом возвращаем его в код, вызвавший функцию `getStartIntent`. Так как в функцию `apply` передаётся функция-расширение для типа-приёмника (T), вызываем его методы напрямую, без обращения к объекту. Сам объект, у которого вызвана функция, будет доступен внутри неё как `this`.

У нас есть доступ не только к методам ресивера, как в случае с `Intent`, но и к переменным. Мы создали экземпляр класса `Person` и определили его переменные, прежде чем присвоить этот класс переменной `peter`.

Эквивалентом будет такая запись:

```
val peter = Person()
clark.name = "Peter"
clark.age = 18
```

## Использование also

`Also` полезен для выполнения дополнительных действий над объектом (не его изменений) и валидации, прежде чем присваивать значение переменной:

```
class Book(author: Person) {
    val author = author.also {
        requireNotNull(it.age)
        print(it.name)
    }
}
```

Тот же вариант, но без `also`:

```
class Book(val author: Person) {
    init {
```

```
        requireNotNull(author.age)
        print(author.name)
    }
}
```

## Использование with

**Важно!** Используем with только на ресиверах non-nullable, и когда не требуется результат выполнения функции:

```
val person: Person = getPerson()
with(person) { //person не должен быть null!
    print(name)
    print(age)
}
```

Без with:

```
val person: Person = getPerson()
print(person.name)
print(person.age)
```

## Использование run

Эта функция применяется, если требуется просчитать какие-то значения или ограничить применение локальных переменных:

```
val inserted: Boolean = run {
    val person: Person = getPerson()
    val personDao: PersonDao = getPersonDao()
    personDao.insert(person)
}
fun printAge(person: Person) = person.run {
    print(age)
}
```

Без run мы создаём переменные, которые доступны внутри метода или всего класса:

```
val person: Person = getPerson()
val personDao: PersonDao = getPersonDao()
val inserted: Boolean = personDao.insert(person)
fun printAge(person: Person) = {
    print(person.age)
}
```

Эта функция наименее популярна из пяти и уступает в популярности `let`. Они похожи, но в `run` нельзя использовать `it`.

## Использование функций в цепочке вызовов

Может возникнуть желание вызывать одну функцию внутри другой, передавая результат операции всё дальше. Старайтесь этого избегать, потому что такой код сложно читать и легко запутаться во множестве `it` и `this`. Вместо использования вложенности, вызывайте эти функции в цепочке, что выглядит чище и приносит преимущества функционального программирования в объектно-ориентированный мир:

```
private fun insert(user: User) = SqlBuilder().apply {
    append("INSERT INTO user (email, name, age) VALUES ")
    append("?", user.email)
    append(",?", user.name)
    append(",?", user.age)
}.also {
    print("Executing SQL update: $it.")
}.run {
    jdbc.update(this) > 0
}
```

Стандартная библиотека Kotlin содержит множество полезных функций-расширений. Нельзя рассмотреть их все в рамках этого курса. Но можно ознакомиться с ними в [документации](#) и посмотреть [исходный код](#).

## Делегирование

Делегирование — это шаблон проектирования, где объект внешне выражает поведение, но на самом деле передаёт ответственность другому объекту.

В Kotlin делегирование реализуется на уровне языка. Это означает, что можно легко делегировать выполнение операций, объявленных в классе, другому объекту. Для реализации такого поведения используется ключевое слово `by`.

В языке Kotlin есть встроенные реализации этой возможности. Наиболее полезная — ленивая инициализация:

```
override val viewModel: MainViewModel by lazy {
    ViewModelProvider(this).get(MainViewModel::class.java)
}
```

Функция `lazy` принимает в качестве параметра функцию, которая возвращает начальное значение свойства. Переданная функция вызовется в момент первого обращения к свойству. Значение, произведённое этой функцией, кешируется и возвратится к запросившему. При последующих обращениях будет возвращаться кешированное значение. Иными словами, наша модель сформируется только тогда, когда к ней впервые обратятся, или не сформируется, если к ней так никто и не обратится. Это экономит ресурсы, так как экземпляры классов создаются, только когда они кому-то нужны. Создание класса — самая ресурсоёмкая операция.

Подробнее о делегировании — на курсе «Профессиональная разработка приложений».

## Хорошие практики при программировании на Kotlin

Выражения в Kotlin возвращают значения и их надо использовать.

Когда хочется сделать так:

```
fun getDefaultLocale(deliveryArea: String): Locale {
    val deliverAreaLower = deliveryArea.toLowerCase()
    if (deliverAreaLower == "germany" || deliverAreaLower == "austria") {
        return Locale.GERMAN
    }
    if (deliverAreaLower == "usa" || deliverAreaLower == "great britain") {
        return Locale.ENGLISH
    }
    if (deliverAreaLower == "france") {
        return Locale.FRENCH
    }
    return Locale.ENGLISH
}
```

Лучше сделать так:

```
fun getDefaultLocale(deliveryArea: String) = when (deliveryArea.toLowerCase()) {
    "germany", "austria" -> Locale.GERMAN
    "usa", "great britain" -> Locale.ENGLISH
    "france" -> Locale.FRENCH
    else -> Locale.ENGLISH
}
```

**Важно!** Каждый раз, когда используется `if`, надо иметь в виду, что часто его можно заменить на более короткую запись посредством `when`.

`try-catch` тоже возвращает значение:

```
val json = """{"message":"HELLO"}"""
val message = try {
    JSONObject(json).getString("message")
} catch (ex: JSONException) {
    json
}
```

В Java мы часто создаём статичные классы со статичными методами для утилит. Не стоит это делать в Kotlin:

```
object StringUtil {
    fun countAmountOfX(string: String): Int{
        return string.length - string.replace("x", "").length
    }
}
StringUtil.countAmountOfX("xXx")
```

Kotlin позволяет убрать ненужные оборачивания в класс. Для этого используются функции верхнего уровня. Мы также можем добавить некоторые функции расширения для повышения читабельности:

```
fun String.countAmountOfX(): Int {
    return length - replace("x", "").length
}
"xXx".countAmountOfX()
```

Для объединения вызовов инициализации объекта используем `apply()`. Часто мы видим такое:

```
val dataSource = BasicDataSource()
dataSource.driverClassName = "com.mysql.jdbc.Driver"
dataSource.url = "jdbc:mysql://domain:3309/db"
dataSource.username = "username"
dataSource.password = "password"
dataSource.maxTotal = 40
dataSource.maxIdle = 40
dataSource.minIdle = 4
```

Функция расширения `apply()` позволяет объединить код инициализации объекта. К тому же нам не требуется повторять название переменной снова и снова:

```
val dataSource = BasicDataSource().apply {
    driverClassName = "com.mysql.jdbc.Driver"
    url = "jdbc:mysql://domain:3309/db"
    username = "username"
    password = "password"
}
```

```
maxTotal = 40
maxIdle = 40
minIdle = 4
}
```

Избегаем if-null-проверок. Java-способ проверки на null громоздкий и позволяет легко пропустить ошибку:

```
if (order == null || order.customer == null || order.customer.address == null) {
    throw IllegalArgumentException("Invalid Order")
}
val city = order.customer.address.city
```

Каждый раз, когда мы пишем проверку на null, важно остановиться. Kotlin предоставляет более простой способ обработки таких ситуаций. Чаще всего используется безопасный вызов «?.» или просто оператор «Элвис» — «?:».

```
val city = order?.customer?.address?.city ?: throw
IllegalArgumentException("Invalid Order")
```

Всё вышесказанное также справедливо и для проверок типов. Не надо использовать is:

```
if (service !is CustomerService) {
    throw IllegalArgumentException("No CustomerService")
}
service.getCustomer()
```

...где можно использовать as?:

```
service as? CustomerService ?: throw IllegalArgumentException("No
CustomerService")
service.getCustomer()
```

Посредством as? и ?: можно проверить тип, автоматически преобразовать его к нужному или бросить исключение, если тип не тот, что ожидается. Всё в одно выражение!

Избегаем вызовов без проверок через «!!». Это плохая практика:

```
order!!.customer!!.address!!.city
```



Использование «!!» смотрится довольно грубо, будто мы кричим на компилятор. Это неслучайно: разработчики языка Kotlin пытаются слегка подтолкнуть нас к поиску лучшего решения, чтобы не использовать выражение, которое не проверяется компилятором.

Используем `let()`. В некоторых ситуациях `let()` позволяет заменить `if`. Но его надо использовать с осторожностью, чтобы код оставался читабельным. Как это делается в стиле Java:

```
val order: Order? = findOrder()
if (order != null) {
    dun(order.customer)
}
```

С `let()` дополнительная переменная не требуется. Поэтому дальше мы имеем дело с одним выражением:

```
findOrder()?.let { dun(it.customer) }
//или ещё короче
findOrder()?.customer?.let (::dun)
```

Если блок кода содержит одну функцию, где `it` — аргумент, то лямбда-выражение обычно заменяется ссылкой на метод `::`.

Можно писать функции, состоящие из одного выражения.

Не надо писать таким образом:

```
fun mapToDTO(entity: SnippetEntity): SnippetDTO {
    val dto = SnippetDTO(
        code = entity.code,
        date = entity.date,
        author = "${entity.author.firstName} ${entity.author.lastName}"
    )
    return dto
}
```

Лучше писать так:

```
fun mapToDTO(entity: SnippetEntity) = SnippetDTO(
    code = entity.code,
    date = entity.date,
    author = "${entity.author.firstName} ${entity.author.lastName}"
)
val dto = mapToDTO(entity)
```

Переменным и функциям разрешается сразу присваивать значения других функций и выражений. Если предпочтительнее функции-расширения, то можно, используя их, одновременно сделать объявление и использование более краткими и выразительными. В то же время мы не загрязняем наши объекты-значения дополнительной логикой:

```
fun SnippetEntity.toDTO() = SnippetDTO(
    code = code,
    date = date,
    author = "${author.firstName} ${author.lastName}"
)
val dto = entity.toDTO()
```

Лучше применять параметры конструктора в инициализации свойств. Важно дважды подумать, прежде чем использовать блок инициализации (init-блок) в теле конструктора, только чтобы инициализировать свойства:

```
class UsersClient(baseUrl: String, appName: String) {
    private val usersUrl: String
    private val httpClient: HttpClient
    init {
        usersUrl = "$baseUrl/users"
        val builder = HttpClientBuilder.create()
        builder.setUserAgent(appName)
        builder.setConnectionTimeToLive(10, TimeUnit.SECONDS)
        httpClient = builder.build()
    }
    fun getUsers() {
        //call service using httpClient and usersUrl
    }
}
```

В инициализации свойств можно ссылаться на параметры основного конструктора — и не только в init-блоке. `apply()` также позволяет сгруппировать код инициализации и обойтись одним выражением:

```
class UsersClient(baseUrl: String, appName: String) {
    private val usersUrl = "$baseUrl/users"
    private val httpClient = HttpClientBuilder.create().apply {
        setUserAgent(appName)
        setConnectionTimeToLive(10, TimeUnit.SECONDS)
    }.build()
    fun getUsers() {
        //call service using httpClient and usersUrl
    }
}
```

Специальные конструкции для создания структур `listOf`, `mapOf` и инфиксная функция `to` используется для быстрого создания структур. Например, JSON:

```
val customer = mapOf(
    "name" to "Clair Grube",
    "age" to 30,
    "languages" to listOf("german", "english"),
    "address" to mapOf(
        "city" to "Leipzig",
        "street" to "Karl-Liebknecht-Straße 1",
        "zipCode" to "04107"
    )
)
```

Правда, обычно для создания JSON приходится использовать `data class` или сопоставление объектов. Но иногда, в том числе и в тестах, такая запись весьма полезна. О формате JSON и конвертации его в дата-класс поговорим позже.

## Практика

Отрефакторим наше приложение. Сделаем его похожим на приложение, написанное на Kotlin, а не на Java. Воспользуемся преимуществами языка.

`RepositoryImpl`, было и стало:

```
class RepositoryImpl : Repository {

    override fun getWeatherFromServer(): Weather {
        return Weather()
    }

    override fun getWeatherFromLocalStorageRus(): List<Weather> {
        return getRussianCities()
    }

    override fun getWeatherFromLocalStorageWorld(): List<Weather> {
        return getWorldCities()
    }
}

class RepositoryImpl : Repository {

    override fun getWeatherFromServer() = Weather()

    override fun getWeatherFromLocalStorageRus() = getRussianCities()
```

```
    override fun getWeatherFromLocalStorageWorld() = getWorldCities()
}
```

Weather.kt:

```
fun getWorldCities() = listOf(
    ...
)

fun getRussianCities() = listOf(
    ...
)
```

DetailsFragment.kt, было и стало:

```
override fun onCreateView(view: View, savedInstanceState: Bundle?) {
    super.onCreateView(view, savedInstanceState)
    val weather = arguments?.getParcelable<Weather>(BUNDLE_EXTRA)
    if (weather != null) {
        val city = weather.city
        binding.cityName.text = city.city
        binding.cityCoordinates.text = String.format(
            getString(R.string.city_coordinates),
            city.lat.toString(),
            city.lon.toString()
        )
        binding.temperatureValue.text = weather.temperature.toString()
        binding.feelsLikeValue.text = weather.feelsLike.toString()
    }
}

override fun onCreateView(view: View, savedInstanceState: Bundle?) {
    super.onCreateView(view, savedInstanceState)
    arguments?.getParcelable<Weather>(BUNDLE_EXTRA)?.let { weather ->
        weather.city.also { city ->
            binding.cityName.text = city.city
            binding.cityCoordinates.text = String.format(
                getString(R.string.city_coordinates),
                city.lat.toString(),
                city.lon.toString()
            )
            binding.temperatureValue.text = weather.temperature.toString()
            binding.feelsLikeValue.text = weather.feelsLike.toString()
        }
    }
}
```

Обратим внимание на использование `let` и `also`: у них явно указываются и даются имена аргументу, когда использование `it` может только всё запутать:

```
arguments?.getParcelable<Weather>(BUNDLE_EXTRA)?.let {
    it.city.also {
        binding.cityName.text = it.city
        binding.cityCoordinates.text = String.format(
            getString(R.string.city_coordinates),
            it.lat.toString(),
            it.lon.toString()
        )
        binding.temperatureValue.text = it.temperature.toString()
        binding.feelsLikeValue.text = it.feelsLike.toString()
    }
}
```

Пользуемся этой особенностью чаще для наглядности кода.

`MainFragmentAdapter.kt`, было и стало:

```
fun bind(weather: Weather) {
    itemView.findViewById<TextView>(R.id.mainFragmentRecyclerItemTextView).text =
weather.city.city
    itemView.setOnClickListener {
        onItemClick?.onItemClick(weather)
    }
}

fun bind(weather: Weather) {
    itemView.apply {
        findViewById<TextView>(R.id.mainFragmentRecyclerItemTextView).text =
weather.city.city
        setOnClickListener { onItemClick?.onItemClick(weather) }
    }
}
```

`MainFragment.kt`, было и стало:

```
private lateinit var viewModel: MainViewModel
private var isDataSetRus: Boolean = true
private val adapter = MainFragmentAdapter(object : OnItemClickListener {
    override fun onItemClick(weather: Weather) {
        val manager = activity?.supportFragmentManager
        if (manager != null) {
            val bundle = Bundle()
            bundle.putParcelable(DetailsFragment.BUNDLE_EXTRA, weather)
            manager.beginTransaction()
                .add(R.id.container, DetailsFragment.newInstance(bundle))
        }
    }
})
```

```

        .addToBackStack("")
        .commitAllowingStateLoss()
    }
}
}))

private val viewModel: MainViewModel by lazy {
    ViewModelProvider(this).get(MainViewModel::class.java) }

private var isDataSetRus: Boolean = true
private val adapter = MainFragmentAdapter(object : OnItemClickListener {
    override fun onItemClick(weather: Weather) {
        activity?.supportFragmentManager?.apply {
            beginTransaction()
                .add(R.id.container, DetailsFragment.newInstance(Bundle()).apply {
                    putParcelable(DetailsFragment.BUNDLE_EXTRA, weather)
                })
                .addToBackStack("")
                .commitAllowingStateLoss()
            }
        }
    })
})

```

Теперь наша `ViewModel` создаётся через ленивую инициализацию, а не в методе `onViewCreated`, а новый фрагмент формируется через «?» и `apply`.

Метод `changeWeatherDataSet` можно написать так. Было и стало:

```

private fun changeWeatherDataSet() {
    if (isDataSetRus) {
        viewModel.getWeatherFromLocalSourceWorld()
        mainFragmentFAB.setImageResource(R.drawable.ic_earth)
    } else {
        viewModel.getWeatherFromLocalSourceRus()
        mainFragmentFAB.setImageResource(R.drawable.ic_russia)
    }
    isDataSetRus = !isDataSetRus
}

private fun changeWeatherDataSet() =
    if (isDataSetRus) {
        viewModel.getWeatherFromLocalSourceWorld()
        mainFragmentFAB.setImageResource(R.drawable.ic_earth)
    } else {
        viewModel.getWeatherFromLocalSourceRus()
        mainFragmentFAB.setImageResource(R.drawable.ic_russia)
    }.also { isDataSetRus = !isDataSetRus }

```

Использование этого метода зависит от того, насколько просто читается код. Иногда сокращения неуместны.

Создадим extension-функцию для Snackbar:

```
private fun View.showSnackBar(  
    text: String,  
    actionText: String,  
    action: (View) -> Unit,  
    length: Int = Snackbar.LENGTH_INDEFINITE  
) {  
    Snackbar.make(this, text, length).setAction(actionText, action).show()  
}
```

Добавим id корневому контейнеру фрагмента:

```
<?xml version="1.0" encoding="utf-8"?>  
<androidx.constraintlayout.widget.ConstraintLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:id="@+id/mainFragmentRootView"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    xmlns:tools="http://schemas.android.com/tools"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
    <androidx.recyclerview.widget.RecyclerView  
        android:id="@+id/mainFragmentRecyclerView"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"  
        app:layout_constraintEnd_toEndOf="parent"  
        app:layout_constraintStart_toStartOf="parent"  
        app:layout_constraintTop_toTopOf="parent" />  
    ...
```

В состоянии ошибки будем вызывать нашу новую функцию у корневого View экрана:

```
is AppState.Error -> {  
    mainFragmentLoadingLayout.visibility = View.GONE  
    mainFragmentRootView.showSnackBar(  
        getString(R.string.error),  
        getString(R.string.reload),  
        { viewModel.getWeatherFromLocalSourceRus() })  
}
```

Теперь наше приложение похоже на то, что написано на Kotlin!

# Факультатив: extension-функции для View

Вот примеры функций-расширений, которые можно использовать в своём приложении. Это часто используемые действия, для которых обычно пишутся статические методы в Java. Но в Kotlin этого можно добиться более красивым и изящным способом.

## Отображать и прятать клавиатуру

```
// Расширяем функционал view для отображения клавиатуры
fun View.showKeyboard() {
    val imm = context.getSystemService(Context.INPUT_METHOD_SERVICE) as
    InputMethodManager
    this.requestFocus()
    imm.showSoftInput(this, 0)
}

// Расширяем функционал view для скрытия клавиатуры
fun View.hideKeyboard(): Boolean {
    try {
        val inputMethodManager =
            context.getSystemService(Context.INPUT_METHOD_SERVICE) as InputMethodManager
        return inputMethodManager.hideSoftInputFromWindow(windowToken, 0)
    } catch (ignored: RuntimeException) { }
    return false
}

// Пример использования
editTextName.showKeyboard()
buttonSubmit.hideKeyboard()
```

## Показать Snackbar

Так как для отображения Snackbar всё равно требуется View, можно добавить такое расширение. Передаём текст сообщения и текст для action, функцию для action — в качестве параметра, и длительность, которая стоит по умолчанию как INDEFINITE, то есть этот параметр можно не передавать:



```
// Ресурс вместо строки
fun View.createAndShow(text: String, actionText: String, action: (View) -> Unit,
length: Int = Snackbar.LENGTH_INDEFINITE) {
    Snackbar.make(this, text, length).setAction(actionText, action).show()
}
```

## Управлять видимостью View

```
fun View.show() : View {
    if (visibility != View.VISIBLE) {
        visibility = View.VISIBLE
    }
    return this
}

fun View.hide() : View {
    if (visibility != View.GONE) {
        visibility = View.GONE
    }
    return this
}

// Отображать вью в зависимости от условия
inline fun View.showIf(condition: () -> Boolean) : View {
    if (visibility != View.VISIBLE && condition()) {
        visibility = View.VISIBLE
    }
    return this
}

inline fun View.hideIf(predicate: () -> Boolean) : View {
    if (visibility != View.GONE && block()) {
        visibility = View.GONE
    }
    return this
}

// Примеры использования
button.hide()
button.show()
button.showIf {
    editTextName.text != null
}
button.hideIf {
    editTextName.text == null
}
```

## Получить Bitmap из View

```
fun View.getBitmap(): Bitmap {  
    val bmp = Bitmap.createBitmap(width, height, Bitmap.Config.ARGB_8888)  
    val canvas = Canvas(bmp)  
    draw(canvas)  
    canvas.save()  
    return bmp  
}  
  
// Пример использования  
val imageBitmap = imageView.getBitmap()
```

## Практическое задание

1. Проведите рефакторинг вашего приложения в соответствии с полученными знаниями о возможностях языка и хорошими практиками программирования на Kotlin.
2. Напишите дополнительные extension-функции для Snackbar без action, а также такие, что принимают строковые ресурсы (R.string...) в качестве текста.

**Задача для самостоятельного изучения:** изучите тему делегатов и делегирования более подробно.

## Дополнительные материалы

1. [Coding convention](#).
2. [Функциональное программирование](#).
3. [Функции-расширения](#).
4. [Функции-расширения](#).

## Используемые источники

1. [Learn Kotlin](#).
2. [Обобщения \(Generics\)](#).

3. [Idiomatic Kotlin. Best Practices.](#)
4. Дмитрий Жемеров, Светлана Исакова «Kotlin в действии».