# 15.KOTLIN. Обощения, карутины. Kotlin для Android

## 15.1 Обощенные типы

Основные идеи объявления и использования обобщенных классов и функций в Kotlin напоминают Java. Однако, в тему обобщенных типов введены новые понятия: *овеществляемые типовые параметры* и *определение вариантности в месте объявления*.

**Овеществляемые типовые параметры** (reified type parameters) позволяют ссылаться во время выполнения на конкретные типы, используемые как типовые аргументы в вызовах встраиваемых функций. (Для обычных классов или функций это невозможно, потому что типовые аргументы стираются во время выполнения.)

**Определение вариантности в месте объявления** (declaration-site variance) позволяет указать, является ли обобщенный тип, имеющий типовой аргумент, подтипом или супертипом другого обобщенного типа с тем же базовым типом и другим типовым аргументом (например, возможность передачи типа List<Int> в функцию, ожидающую List<Any>).

Поддержка обобщенных типов дает возможность определять *параметризованные типы*. Когда создается экземпляр такого типа, параметр типа заменяется конкретным типом, который называется **типовым аргументом.** Синтаксис выглядит в точности как в Java.

*Обобщенные функции и свойства*

**Типовые параметры можно объявлять для методов классов или интерфейсов, функций верхнего уровня и функций-расширений**.

Обобщенная функция имеет свои типовые параметры. Такие типовые параметры замещаются конкретными типовыми аргументами во всех вызовах функции. Большая часть библиотечных функций, работающих с коллекциями, - обобщенные.

Ниже определяется свойство-расширение, которое возвращает предпоследний элемент в списке

```
val <T> List<T>.pred: T
    get() = this[size - 2]

fun main (){

    println(listOf(1, 2, 3, 4).pred)
}
```

Обычные свойства (не являющиеся расширениями) не могут иметь типовых параметров - нельзя сохранить несколько значений разных типов в свойстве класса, а поэтому не имеет смысла объявлять свойства обобщенного типа, не являющиеся расширениями.

*Обощенные классы*

В Kotlin имеется возможность объявлять обобщенные классы или интерфейсы, поместив угловые скобки после имени класса и указав в них типовые параметры.

```
interface SomeList<T> {
    operator fun get(index: Int): T
//...
}
```

Если ваш класс наследует обобщенный класс (или реализует обобщенный интерфейс), вы должны указать типовой аргумент для обобщенного параметра базового типа. Это может быть или конкретный тип, или другой типовой параметр

```
class TList <T> :SomeList <T> {
override fun get(index: Int): T = ...
}
```

В качестве типового аргумента для класса может даже использоваться сам класс.

```
class StringList :SomeList <String> {
override fun get(index: Int): String = ...
}
```

*Ограничения типовых параметров*

Ограничения типовых параметров позволяют ограничивать круг допустимых типов, которые можно использовать в качестве типовых аргументов для классов и функций.

Чтобы определить ограничение, нужно добавить двоеточие после имени типового параметра, а затем указать тип, который должен стать **верхней границей**. Когда какой-то тип определяется как верхняя граница для типового параметра, в качестве соответствующих типовых аргументов должен указываться либо именно этот тип, либо его подтипы

```
fun <T : Number> oneHalf(value: T): Double {
return value.toDouble() /2.0
}
```

Когда объявляется обобщенный класс или функция, вместо его типового параметра может быть подставлен любой типовой аргумент, включая типы с поддержкой значения null. То есть типовой параметр без верхней границы на деле имеет верхнюю границу Any?.

Чтобы гарантировать замену типового параметра только типами, не поддерживающими значения null, необходимо объявить ограничение.

```
class Processors  <T: Any> {
    fun process(value: T) {
        value.hashCode()
    }
}
```

Обратите внимание, что такое ограничение можно наложить указанием в верхней границе любого типа, не поддерживающего null, не только типа Any.

В редких случаях, когда требуется определить несколько ограничений для типового параметра, можно использовать другой синтаксис

```
fun <T> restriction(seq: T)
    where T : CharSequence, T : Appendable {
    //...
}
```

В данном случае указали, что тип, определенный в качестве типового аргумента, должен реализовать два интерфейса.

**Обобщенные типы во время выполнения: стирание и овеществление параметров типов**

Обобщенные типы реализуются в JVM через механизм стирания типа (type erasure) - то есть типовые аргументы экземпляров обобщенных классов не сохраняются во время выполнения. Так как типовые аргументы не сохраняются, вы не сможете проверить их - например, вы не сможете убедиться, что конкретный список состоит из строк, а не каких-то других объектов

```
if (value is List<String>) (...)
```

Во время выполнения возможно определить, что значение имеет тип List, но нельзя определить, что список хранит строки, объекты класса Person или что-то ещё, - эта информация стерта.

Чтобы проверить, что значение является списком, а не множеством или каким-то другим объектом, можно сделать с помощью специального синтаксиса *проекций*:

```
if (value is List<*>) (...)
```

Проверяем, является ли value списком List, но не делаем никаких предположений о типе элементов.

Чтобы типовые аргументы не стирались (или, говоря терминами языка Kotlin, овеществлялись), функцию можно объявить встраиваемой.

Объявление встраиваемой функции может способствовать увеличению производительности, особенно если функция принимает аргументы с лямбда-выражениями: код лямбда-выражений также может встраиваться, поэтому отпадает необходимость создавать анонимные классы.

*Здесть используется другое приемущество - типовые параметры встраиваемых функций могут овеществляться - то есть во время выполнения можно ссылаться на фактические типовые аргументы.*

В теле обобщенной функции нельзя ссылаться на типовой аргумент, с которым она была вызвана

```
fun <T> isA(value: Any) = value is T
```

Unresolved reference: T

Если объявить функцию isA с модификатором inline и отметить типовой параметр как овеществляемый (reified):

```
inline fun <reified T> isA(value: Any) = value is T

fun main()
 {
     println(isA<String>("abc"))
     println(isA<Int>(123))
 }
```

Появится возможность проверить, является ли value экземпляром типа T.

Для встраиваемых функций компилятор вставляет байт-код с их реализацией в точки вызова. Каждый раз, когда в программе встречается вызов функции с овеществляемым типовым параметром, компилятор точно знает, какой тип используется в качестве типового аргумента для данного конкретного вызова

Так как байт-код ссылается на конкретный класс, а не на типовой параметр, типовой аргумент не стирается во время выполнения. *Встраиваемые функции с овеществляемыми типовыми параметрами не могут быть вызваны из Java.* Обычные встраиваемые функции можно вызвать из Java.

Встраиваемая функция может иметь несколько овеществляемых типовых параметров, а также неовеществляемые типовые параметры в дополнение к овеществляемым.

Овеществляемые типовые параметры имеют  ограничения.

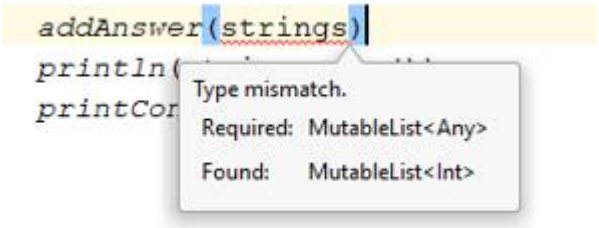
***Вариантность: обобщенные типы и подтипы***

Термин вариантность (variance) описывает, как связаны между собой типы с одним базовым типом и разными типовыми аргументами.

Используется при создании API.

```
fun printContents(list: MutableList<Any>) {
    println(list.toString())
}
fun addAnswer(list: MutableList<Any>) {
    list.add("yes")
}
 fun main (){
     val strings = mutableListOf(1, 2)

     addAnswer(strings)
     println(strings.max())
     printContents (strings)
 }
```




Этот пример показывает, что не всегда безопасно передавать MutableList<String> в аргументе, когда ожидается MutableList<Any>, - компилятор Kotlin справедливо отвергает такие попытки.

Обобщенный класс - например, MutableList - *называют инвариантным по типовому параметру*, если для любых разных типов A и B MutableList<A> не является подтипом или супертипом MutableList<B>. В Java все классы инвариантны

(хотя конкретные декларации, использующие эти классы, могут не быть инвариантными, как вы вскоре увидите).

**Ковариантный класс** - это обобщенный класс (в качестве примера будем использовать Producer<T>), для которого верно следующее: Producer<A> - это подтип Producer<B>, если A - подтип B. Это сохранение направления отношения тип-подтип. Например, Producer<Cat> - это подтип Producer<Animal>, потому что Cat - подтип Animal.

Обозначается

```
interface Producer <out T> { // ковариантный
    fun produce(): T
}
```

Объявление типового параметра класса ковариантным разрешает передавать значения этого класса в аргументах и возвращать их из функций, когда типовой аргумент неточно соответствует типу параметра в определении функции.

Объявление класса ковариантным по определенному типовому параметру ограничивает возможные способы использования этого параметра в классе. Чтобы гарантировать безопасность типов, он может использоваться только в так называемых *исходящих (out) позициях*: то есть класс может производить значения типа T, но не потреблять их. Использование типового параметра в объявлениях членов класса можно разделить *на входящие (in)* и исходящие (out) позиции.

Вариантность защищает от ошибок, когда экземпляры класса используются как экземпляры более обобщенного типа: вы просто не сможете вызвать потенциально опасных методов.

Понятие **контравариантности** можно рассматривать как обратное понятию ковариантности: для контравариантного класса отношение тип-подтип действует в противоположном направлении относительно отношения между классами, использованными как типовые аргументы.

```
interface Comparator<in T> {
    fun compare(e1: T, e2: T): Int {     }
        //...
    }
```

Реализация интерфейса Comparator для данного типа может сравнивать значения любых его подтипов.

Класс, контравариантный по типовому параметру, - это обобщенный класс (возьмем для примера Consumer<T>), для которого выполняется следующее: Consumer<A> - это подтип Consumer<B>, если B - это подтип A. Типовые аргументы A и B меняются местами, поэтому мы говорим, что отношение тип-подтип обратно направленное. Например, Consumer<Animal> - это подтип Consumer<Cat>.

Для класса Producer отношение тип-подтип повторяет соответствующее отношение для типовых аргументов, в то время как для класса Consumer отношения противоположны.


Далее перечислены основные различия между возможными видами вариантности.

| Ковариантные | Контравариантные | Инвариантные |
|---|---|---|
| `Producer<out T>` | `Consumer<in T>` | `MutableList<T>` |
| Направление отношения тип-подтип для классов сохраняется: `Producer<Cat>` - подтип `Producer<Animal>`. | Направление отношения тип-подтип меняется на обратное: `Consumer<Animal>` - подтип `Consumer<Cat>`. | Нет отношения тип-подтип |
| T только в исходящих позициях | T только во входящих позициях | T в любых позициях |

**В Kotlin поддерживается возможность применения и написания аннотаций, механизм рефлекции**

## 15.2 Корутины

Android-приложения могут выполнять самые разные функции. Например, может понадобиться, чтобы приложение скачивало данные, обращалось к базам данных или к веб-службам. Это полезные операции, но они требуют времени для выполнения. Очень нежелательно, чтобы пользователь сидел и ждал, пока приложение завершит операцию, прежде чем сможет продолжить работу с ним.

**Сопрограммы, или корутины**, позволяют выполнять операции в фоновом режиме, или асинхронно.

Надо подключить

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.4.1"
```

Один из способов создать сопрограмму — использовать функцию async из библиотеки поддержки сопрограмм. Функция async принимает один аргумент: лямбду, которая определяет операции для выполнения в фоновом режиме.

```
fun getData() :Deferred<….> {
    return async {
```

```
            URL(CHARACTER_DATA_API).readText()
             return…
}
}
```

Рассмотрим еще пример

```
GlobalScope.launch {
    delay(1000L)
    println("World!")
}
```

launch - это билдер корутины, билдер упакует переданный ему блок кода в корутину и запустит ее. Корутину можно представить так:

```
executor.submit {
    delay(1000L)
    println("World!")
}
```

Это экзекьютор, который оборачивает блок кода в Runnable и запускает этот Runnable на выполнение.

Функции async и launch, которые были использованы называются функциями-строителями сопрограмм (coroutine builder). Это функции, которые настраивают сопрограмму для определенной работы.

*launch создает и сразу запускает сопрограмму*, которая выполняет указанную работу.

Строитель *сопрограммы async* работает иначе, создавая сопрограмму, которая возвращает тип Deferred, обозначающий еще не завершенную работу. Тип Deferred обещает, что работа будет выполнена в какой-то момент в будущем.

Тип Deferred имеет функцию с именем await, которая вызывается тогда, когда нужно дождаться завершения работы и получить результат. await приостанавливает выполнение следующей задачи (обновление UI) до тех пор, пока Deferred не завершит работу.

https://kotlinlang.org/docs/tutorials/coroutines/coroutines-basic-jvm.html#lets-run-a-lot-of-them

Корутина имеет ряд особенностей. Например, когда мы запускаем корутину, мы можем получить результат запуска билдера

```
val job = GlobalScope.launch {
    delay(1000L)
    println("World!")
}
```

Мы можем сделать выполнение кода отложенным (LAZY) и стартовать его позже, когда понадобится. Или в любой момент времени можно будет отменить выполнение. Также можно запускать корутину внутри корутины. Их работы будут связаны между собой отношениями Parent-Child, что является отдельным механизмом, который влияет на обработку ошибок и отмену корутин.

Карутину можно считать легковесным потоком. Как и потоки, сопрограммы могут работать параллельно, ждать друг друга и общаться. Самая большая разница в том, что карутины почти бесплатные: можем создать их тысячи и с точки зрения производительности это будет не заметно.

По умолчанию курутины выполняются в общем пуле потоков. Потоки все еще существуют в программе, основанной на карутинах, но один поток может выполнять много сопрограмм.

Для того чтобы это проверить запустите потоки

```
val c = AtomicLong()

for (i in 1..1_000_000L)
thread(start = true) {
    c.addAndGet(i)
}
println(c.get())
```

затем сделайте то же самое но с карутинами

```
val c = AtomicLong()

for (i in 1..1_000_000L)
GlobalScope.launch {
    c.addAndGet(i)
}
println(c.get())
```

и для того чтобы проверь что они запускаются параллельно добавьте в цикл задержку на 1 секунду

```
val c = AtomicLong()

for (i in 1..1_000_000L)
GlobalScope.launch {
    delay(100)
    c.addAndGet(i)
}

println(c.get())
```

число будет меньше.

Чтобы программа не закнчилась раньше добавьте

```
Thread.sleep(5000) // wait for 5 seconds
println("Stop")
```

**Suspend функция**

Suspend функции не блокирует поток. А значит мы можем эту корутину запускать в main потоке.

Функции приостановки (suspend) разрешено вызывать только из сопрограммы или другой suspend функции .

Например, генерируем URL файла и передаем его в функцию download, которая выполнит загрузку. После этого выводим Toast-сообщение о том, что файл загружен.

Функция download - синхронная, а значит заблокирует поток, в котором будет выполнена. Но хотелось бы выполнить этот код в UI потоке, потому что функция toast может быть выполнена только в нем.

```
    val url = buildUrl()
// long synchronous function
                download(url)
        toast("File is downloaded")
```

Нужно сделать функцию download асинхронной и toast поместить в колбэк, который будет выполнен по завершении загрузки

```
 val url = buildUrl()

// long asynchronous function
            download(url) {
```

```
            toast("File is downloaded")
        }
```

Если функция download синхронна - то поток будет заблокирован. А если она асинхронна, то нам придется добавлять колбэк, чтобы выполнить последующий код.

Для того, чтобы долго работающая функция не блокировала поток, но при этом код, который находится после функции, был выполнен по завершении загрузки без всяких колбэков можно использовать suspend функций в корутинах

Перепишем код с использованием корутины и suspend функции:

```
launch {
    val url = buildUrl()
    download(url) //suspend function
    toast("File is downloaded")
}
```

Эта корутина не заблокирует поток, в котором будет запущен ее код. Функция download загрузит файл в отдельном потоке, а toast будет выполнен позже.

То, что в Kotlin выглядит как корутин и suspend функция, в Java реализовано обычным кодом в обычном классе. Для корутины будет создан отдельный Continuation класс. Именно в этом классе и содержится механизм приостановки корутины suspend функцией.

```
int label;

void invokeSuspend() {
    switch (label) {
        case 0: {
        url = buildUrl();
        label = 1;

        download(url); // suspend function
        return;
    }
        case 1: {
        toast("File is downloaded: " + url);
        return;
    }
    }
}
```

В метод invokeSuspend переехал весь код, который корутина должна выполнить. Этот метод будет вызван при старте корутины. Основная задача Continuation - сделать так, чтобы toast был выполнен только после того, как функция download в фоновом потоке загрузит файл. Обычно в Java в таких случаях используется колбэк, но Continuation в методе invokeSuspend код делится свитчем на две части. и добавляется переменная label.

Точка разделения кода на две части - это suspend функция. Она и весь код перед ней попадет в первую часть. А весь код после нее - во вторую.

Теперь при вызове метода invokeSuspend от переменной label зависит, какая из двух частей будет выполнена. Если label = 0, то выполнится первая часть кода (buildUrl + download). А если label = 1, то вторая (toast). Это дает возможность отделить вызов download от toast.

Получается следующая схема взаимодействия Continuation и suspend функций. Метод invokeSuspend вызывает suspend функции, передает туда Continuation и завершает свою работу. А suspend функции по завершении своей работы снова вызывают invokeSuspend и возвращают поток выполнения кода в блок кода, который в корутине шел после этой функции. При этом они передают туда же результаты своей работы.

*Создание suspend функции*

```
suspend fun download(url: String): File {

}
```

Ключевое слово здесь - suspend. Это даст Котлину понять, что это suspend функция и она будет приостанавливать корутину.

У нас есть некий NetworkService, который асинхронно умеет загружать файлы.

Suspend функция должна результаты своей работы передать в Continuation.invokeSuspend. Для этого используется метод Continuation.resume

```
suspend fun download(url: String): File {
    return suspendCoroutine { continuation ->
        networkService.download(url, object: Net-
workService.Callback {
            override fun onSuccess(result: File) {
                continuation.resume(result)
            }
        })
    }
}
```

Весь наш код уходит в блок suspendCoroutine. Этот блок дает нам доступ к continuation, в который мы передаем результат работы networkService.download.

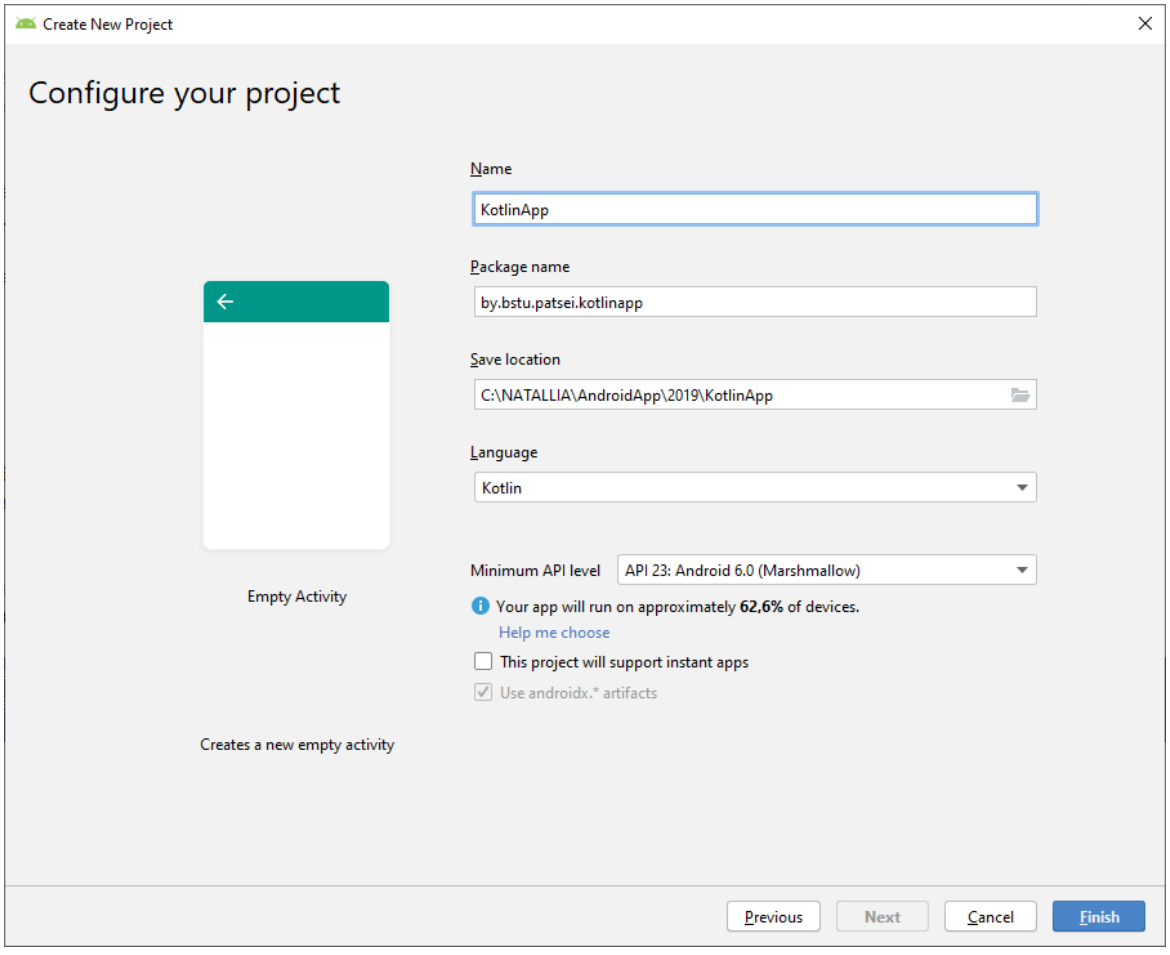
Получается, что suspend функция выполнила асинхронную работу и результат передала в continuation.resume, а тот уже передаст его в continuation.invokeSuspend.

Слово suspend не сделает так, чтобы синхронный код вдруг перестал блокировать поток. Это просто маркер того, что данная функция умеет (и должна) работать с Continuation, чтобы приостановить выполнение корутины не блокируя поток.

Про корутины говорят, что они приостанавливаются, а про традиционный поток — что он «блокируется». Разница в терминологии подсказывает, почему они обеспечивают лучшую производительность, чем потоки: заблокированный поток не может выполнять другую работу, пока не разблокируется. Корутина запускается в потоке — например, в UI-потоке Android или в потоке из пула, но не блокирует его. Поток, выполняющий приостанавливаемую функцию, может выполнять другие сопрограммы. Вот почему корутины обеспечивает более высокую производительность.

https://kotlinlang.org/docs/tutorials/coroutines/coroutines-basic-jvm.html

## 15.3Kotlin для Android



Плагин Kotlin Gradle, позволяет системе Gradle компилировать файлы Kotlin. Стандартная библиотека Kotlin также включена в список зависимостей

```
apply plugin: 'kotlin-android'

apply plugin: 'kotlin-android-extensions'


...
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
```

Класс Activity

```
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Сделаем поиск визуального элемента, поменяем свойства

```
val greetingTextView = findViewById<TextView>(R.id.greet)
greetingTextView.text = "Kotlin application"
greetingTextView.setText("Application in Kotlin")
```

Так как взаимодействуем с Java-классом TextView из Kotlin, Kotlin транслирует соглашение об использовании методов чтения/записи в свой эквивалент этого соглашения — использование синтаксиса обращения к свойствам.

**Синтетические свойства**

Благодаря включению в проект модуля kotlin-adnroid-extensions, допускается использование **синтетических свойств**, которые открывают доступ к визуальным элементам по их атрибутам id. Эти свойства создаются для всех виджетов, объявленных в файле макета.

Расширения Kotlin для Android — это комплект дополнительных функций, включаемых по умолчанию в новый проект с помощью Gradle. Строка кода

```
import kotlinx.android.synthetic.main.activity_main.*
```

благодаря подключаемому модулю kotlin-android-extensions, добавляет последовательность свойств-расширений в Activity. Синтетические свойства сильно упрощают код поиска визуальных элементов, так как больше не нужно вызывать findViewById. Раньше ссылки на визуальные элементы сохранялись в локальных переменных в функции onCreate, теперь у вас есть свойства с именами, соответствующими атрибутам id, объявленным в файле макета.

Например,

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://sche-
mas.android.com/apk/res/android"
    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"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/greet"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Чтобы получить этот компонент, нужно записать его id

```
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

   //     val greetingTextView = findViewById<TextView>(R.id.greet)
//        greetingTextView.text = "Kotlin application"
//        greetingTextView.setText("Application in Kotlin")

            greet.text = "Expent Kotlin"
    }
}
```

Добавим обработчик

```
    greet.setOnClickListener {
     greet.text = "By-by"
}
```

Интерфейс OnClickLestener объявляет только один абстрактный метод — onClick. Подобные интерфейсы называют типами с единственным абстрактным методом (Single Abstract Method, SAM).

Kotlin поддерживает возможность преобразования типов с единственным абстрактным методом, которая позволяет передать в аргументе литерал функции вместо анонимного встроенного класса. Для взаимодействий с кодом на Java, требующим аргумента с реализацией SAM-интерфейса, Kotlin позволяет вместо традиционных анонимных встроенных классов использовать литералы функций.

Заглянув в байт-код этого обработчика, вы увидите, что в нем используется анонимный встроенный класс, как в классическом коде Java.

**С ноября 20-го года синтетические свойства Котлина не рекомендуются.**

**Добавить материал с курса видео Couritne in dephth**

**Gпроеты KotlinLang? KotlinwithCorunit**