GeekBrains

Android на Kotlin

Введение в Kotlin

На этом уроке

1. Разберёмся в базовом синтаксисе языка Kotlin.
2. Научимся настраивать Android Studio для работы с Kotlin.

Оглавление

Преимущества языка Kotlin

Kotlin получил признание в разработке приложений под Android ещё до релиза версии 1.0 в феврале 2016 года. Специалисты ценят его за краткость, выразительность и безопасность. После заявления Google о полной поддержке языка для разработки Android-приложений его популярность росла экспоненциально. Сейчас Kotlin используется для разработки практически 100% новых приложений. Благодаря полной взаимозаменяемости с Java, он подключается и к действующим проектам, если планируется их долгосрочная поддержка. Поэтому Kotlin сегодня — отраслевой стандарт в Android-разработке.

Краткость

Важное преимущество языка — краткость. На Kotlin можно писать те же вещи, что и на Java, но обходиться значительно меньшим количеством кода. Например:

```
public class Person {

    private String firstName;
    private String lastName;
    private int age;

    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public int getAge() {
        return age;
    }
}
```

Это POJO — Plain Old Java Object — класс, который используется в Java для представления простого объекта. То же, только на Kotlin:

```
data class Person(val firstName: String, val lastName: String, val age: Int)
```

Заметим, что в data-классе сразу переопределены методы hashCode() и equals(). В Java-коде это потребовало бы дополнительных усилий. В Kotlin очень просто создавать Parcelable-классы без написания громоздкого кода, как в Java.

Безопасность

Как и Java, Kotlin — это язык со статической типизацией. Поэтому синтаксические ошибки и баги, связанные с неправильным обращением к объектам, отлавливаются во время компиляции. Благодаря этому, оперативно выявляется неработоспособный код, определяются и исправляются ошибки. В отличие от Java, Kotlin позволяет записывать многие вещи так же кратко, как языки с динамической типизацией. Например, так:

```
val name = "Николай"
```

Здесь мы создали новую переменную типа String, но так как сразу присвоили ей значение, нет необходимости обозначать её тип явно. Компилятор способен сам вывести его исходя из значения.

Рассмотрим ещё один аспект безопасности Kotlin. В языке есть концепция nullable types, которая позволяет избежать ошибки NullPointerException. Обычно отловить её можно только во время исполнения программы, а Kotlin даёт сделать это в момент компиляции. Рассмотрим пример:

```
class Person {
   Preson(String firstName, String secondName, int age) {...}
}

Person person = new Person(null, null, 29);
person.getFirstName().length()
```

При выполнении этого кода программа «упадёт» с ошибкой. И узнаем мы об этом только во время исполнения. Конечно, это упрощённый вариант кода, где ошибку видно сразу. Но в реальной программе обнаружить такие баги бывает непросто. Что предлагает Kotlin:

```
class Person(firstName : String, secondName : String, age : Int)
val person = Person(null, null, 29)  // ошибка компиляции
```

В Kotlin с его nullable-типами такой код попросту не скомпилируется. У класса Person в конструкторе заявлены параметры типов, не поддерживающих null в качестве значения. Поэтому компилятор может это отследить и не дать собрать такой код. Подробнее об этом мы поговорим на следующем занятии.

Полная совместимость с Java

Kotlin спроектирован так, чтобы его можно было применять везде, где используется Java: они полностью совместимы. Одна часть кода приложения применяется на Java, а другая — на Kotlin, и всё будет прекрасно работать. Появится возможность обращаться к классам, написанным на Kotlin, из Java-кода и, наоборот, без ограничений.

Возможности современных языков при разработке под Android

При разработке Android-приложений используется Java не выше восьмой версии, где недоступны многие нововведения. Android Studio 3.0 позволяет использовать некоторые вещи из Java 8, но далеко не все её фичи. В Kotlin же есть все эти инструменты и даже больше.

История создания Kotlin

Разработку Kotlin в 2010 году начали специалисты компании JetBrains. Их перестали устраивать возможности Java, и они решили создать свой язык программирования, объединяющий лучшие черты действующих ЯП. Языку дали название Kotlin в честь острова в Финском заливе, где расположен город Кронштадт.

В 2011 году разработчики представили публике новый язык, а в 2012-м открыли его исходный код. Всё это время велись интенсивная разработка и тестирование. В 2016-м появилась релизная версия 1.0. Уже тогда многие Android-разработчики использовали Kotlin в своих проектах. После объявления на Google I/O 2017 о полной поддержке языка для создания мобильных приложений Kotlin стал чрезвычайно популярен в профессиональном сообществе.

Учитывая десятилетнюю историю разработки, Kotlin — это новый язык программирования, и его ставят в один ряд с такими языками как Golang и Rust, в которых содержатся похожие принципы написания кода. Важно иметь это в виду, если потребуется изучить какой-то новый язык.

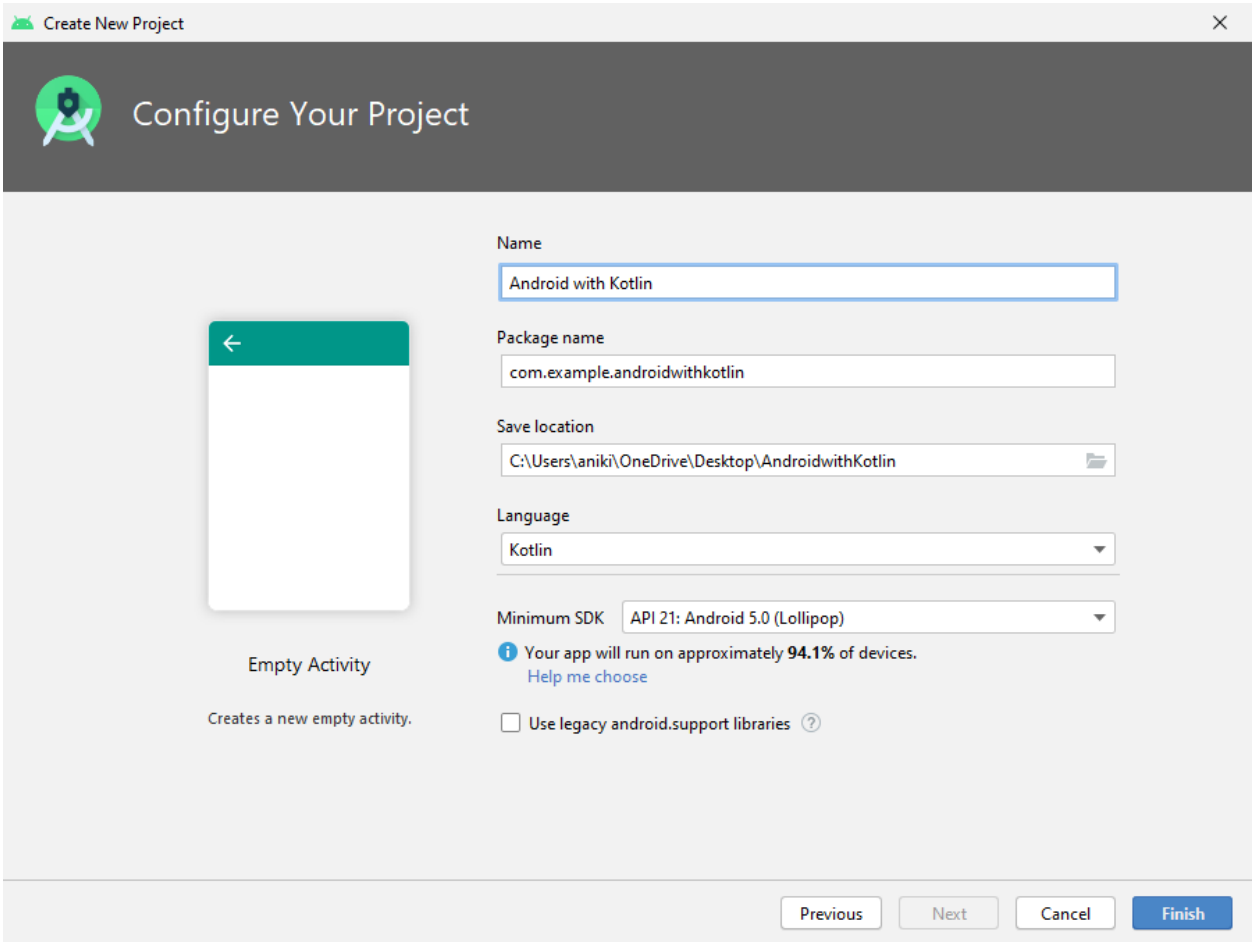
Практика

В практической части урока научимся настраивать Android Studio для работы с Kotlin: создадим на нём новый проект и запустим первое приложение.

Настройка Android Studio

Чтобы настроить Android Studio для работы с Kotlin, подключим к проекту необходимые библиотеки и установим Kotlin-плагин. Самый простой способ это сделать — выбрать Kotlin при создании нового проекта. В Android Studio версии 3.0 и выше он стоит по умолчанию.

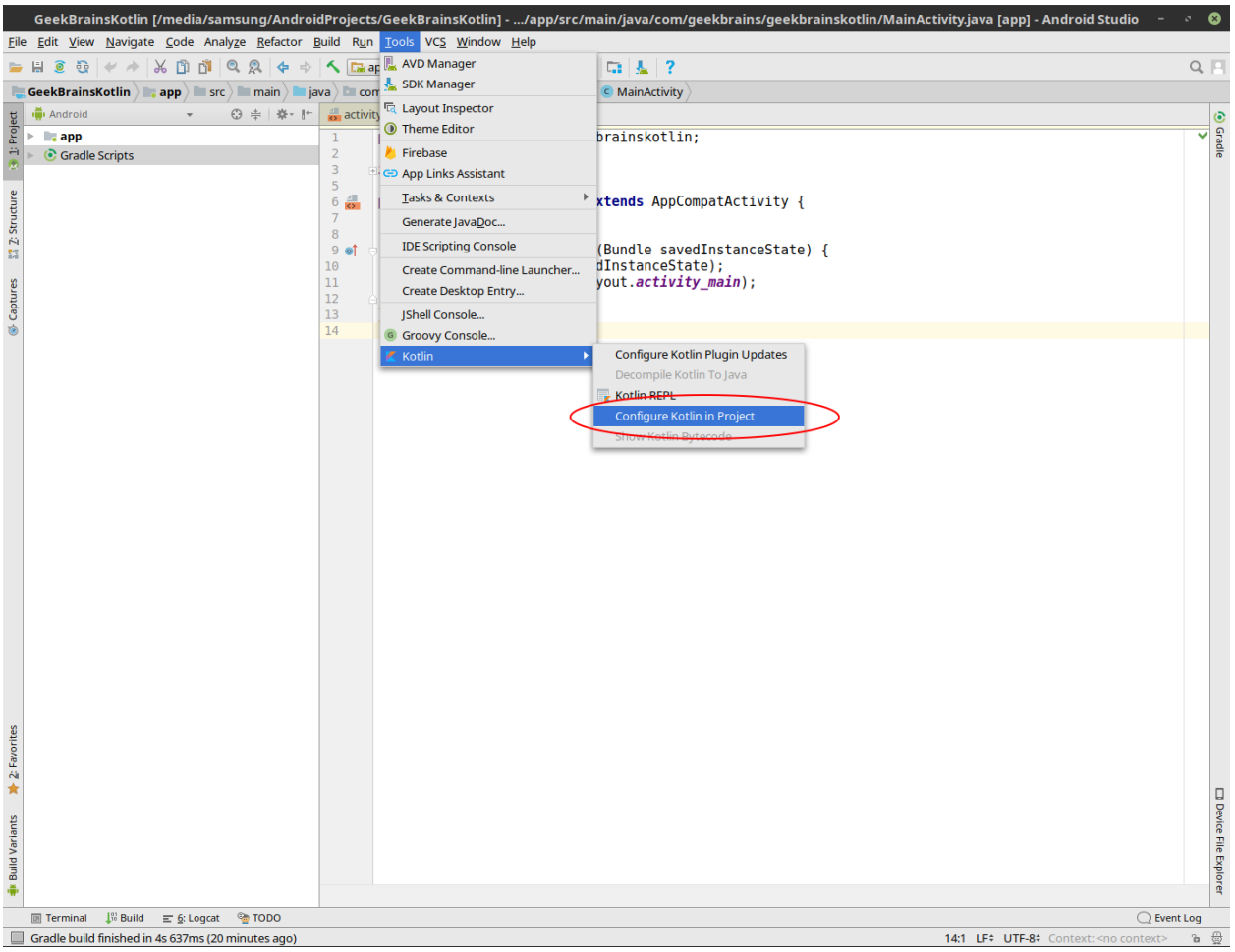
Подключение Kotlin при создании нового проекта:



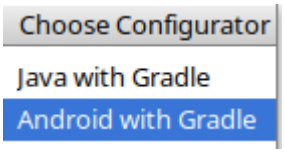
Здесь всё просто. Kotlin стоит по умолчанию, и если просто нажать Finish и создать проект, то в нём сразу подключится всё необходимое для работы с Kotlin. Но иногда требуется подключить Kotlin к проекту, написанному на Java.

Подключение Kotlin к действующему проекту

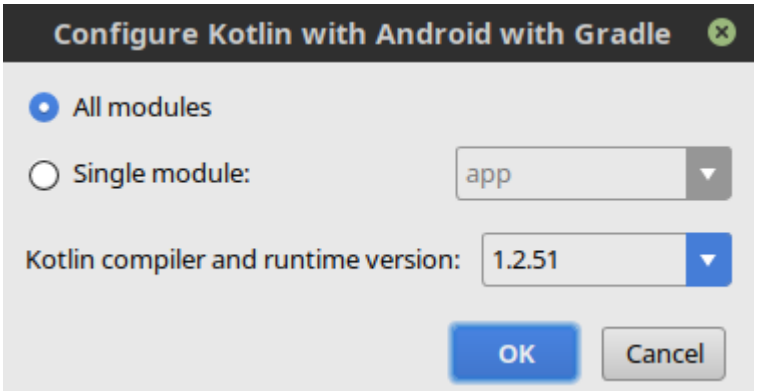
В меню Android Studio выбираем Tools → Kotlin → Configure Kotlin in project:



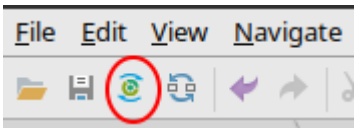
Далее в выпадающем окне выбираем Android with Gradle:



И после этого оставляем отмеченным All modules. Чтобы настроить Kotlin только в отдельных модулях, указываем, в каких:



Нажимаем OK и синхронизируем Gradle:



Kotlin сконфигурирован в проекте.

Заметим, что в проекте отдельно прописываются следующие данные в файлах Gradle: отдельная переменная для версии языка и плагин для Gradle:

```
// Top-level build file where you can add configuration options common to all sub
buildscript {
    ext.kotlin_version = "1.3.72"
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath "com.android.tools.build:gradle:4.0.0"
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

Плагин для Gradle (Kotlin и расширения) и две зависимости в самом низу:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

android {
    compileSdkVersion 30
    buildToolsVersion "30.0.0"

    defaultConfig {
        applicationId "com.example.androidwithkotlin"
        minSdkVersion 21
        targetSdkVersion 30
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.tx
        }
    }
}

dependencies {
    implementation fileTree(dir: "libs", include: ["*.jar"])
    implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
    implementation 'androidx.core:core-ktx:1.3.0'
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

Перевод Java-классов в Kotlin средствами Android Studio

В Java-проекте есть возможность автоматически переводить классы с Java на Kotlin. Требуется просто кликнуть правой кнопкой мыши по нужному файлу в Android Studio и в выпадающем меню выбрать Convert Java file to Kotlin File. Далее Android Studio сделает всё сама, и мы увидим тот же файл, только написанный на Kotlin. Важно быть осторожными и перепроверять всё после такой конвертации: как и все роботизированные функции, конвертация происходит не всегда корректно или оптимально.

Классы в Kotlin

Объявление классов

Как и в Java, для объявления классов в Kotlin используется ключевое слово class. Далее следует конструктор, обозначенный соответствующим ключевым словом:

```
class Weather constructor(var town: String, var temperature: Int)
```

Это пример объявления класса Weather, который имеет конструктор, принимающий в качестве параметров строку и Int. Заметим: у класса нет тела. В Kotlin это допустимо, если задача класса — только содержать данные. В Java мы привыкли в теле конструктора присваивать переданные параметры переменным класса. В языке Kotlin за это отвечает ключевое слово var, о котором поговорим позже.

Ключевое слово constructor обычно опускается:

```
class Weather(var town: String, var temperature: Int)
```

Часто требуется выполнять первичную инициализацию объекта при его создании. В Java это обычно делается в конструкторе. В Kotlin для этого есть блок Init, который будет выполняться при создании объекта:

```
class Repository {

   private val weatherList: List<Weather>

   init {
       weatherList = listOf( Weather("Москва", 15))
   }
}
```

Здесь в момент создания класса инициализируем свойство класса weatherList списком из одного объекта Weather. Ещё в этом примере есть вызов конструктора класса Weather. Заметим, что в Kotlin нет ключевого слова new. listOf — это функция из стандартной библиотеки. Об этой и других полезных функциях поговорим позже.

Конечно же, инициализировать переменные можно не только в блоке init, но и напрямую при объявлении переменной.

Наследование

В Java родитель всех классов — Object, а в Kotlin все классы наследуются от Any. Чтобы унаследоваться от другого класса, надо сделать следующее:

```
class WeatherViewHolder(itemViev: View) : RecyclerView.ViewHolder(itemViev)
```

Имя родительского класса указывается после двоеточия, как и имя реализуемого интерфейса. Если интерфейсов несколько, то они перечисляются через запятую. В этом случае родительский класс имеет конструктор, принимающий View в качестве параметра, поэтому мы обязаны передать его при создании.

Чтобы проинициализировать базовый класс, WeatherViewHolder должен получить View при создании и передать её в конструктор базового класса. Запись, приведённая в примере выше, приравнивается вызову super(itemView) в Java. Java-классы наследуются так же, как и в Java, если они не объявлены final. В Kotlin все классы по умолчанию закрыты для наследования — final в терминах Java. Чтобы создавать наследников класса, его надо объявить с ключевым словом open:

```
open class BaseViewModel : ViewModel() {
}
```

Теперь вынесем в тело этого класса общий для всех инструментарий ViewModel и используем этот класс как родителя для всех наших ViewModel. Когда конструктор родительского класса не принимает параметров, мы всё равно должны его вызвать (пустые скобки). **Пустой конструктор создаваемого класса можно опустить.**

Синглтон

В Kotlin есть удобный и простой способ создавать синглтоны. Чтобы класс всегда оставался в единственном экземпляре на протяжении работы приложения, достаточно заменить слово class на object:

```
object Repository {
    private val weatherList: List<Weather>

    fun getWeatherList(): List<Weather> {
        return weatherList
     }
}
```

Определённый таким образом класс будет синглтоном. Обращаться из кода на Kotlin к нему можно напрямую:

```
Repository.getWeatherList()
```

Если надо вызвать этот класс из Java, то писать потребуется немного больше. Это специфический синтаксис взаимозаменяемых языков. Об INSTANCE — далее.

```
Repository.INSTANCE.getWeatherList()
```

Функции

Функции в Kotlin объявляются следующим образом:

```
fun getWeather(): List<Weather> {
   return weatherList
}
```

Первым следует ключевое слово fun, далее — название функции с принимаемыми аргументами в скобках, после двоеточия — возвращаемое значение. После объявления идёт тело функции в фигурных скобках. Когда функция ничего не возвращает, возвращаемое значение опускается:

```
fun onBindViewHolder(holder: WeatherViewHolder, position: Int) {
   holder.bind(weatherList[position])
}
```

Возвращаемое значение приведённой выше функции — Unit. Это аналог void в Java, только Unit — это объект. Об этом и других типах в Kotlin поговорим позже. Когда тело функции записывается одним выражением, фигурные скобки можно опустить и записать через «=». Да, к этому тоже надо привыкнуть.

```
override fun getItemCount() = weatherList.size
```

В нашем случае компилятор может вывести тип из возвращаемого значения выражения, поэтому указывать его необязательно. Ключевое слово override в начале объявления функции — аналог аннотации @Override в Java. Это означает, что такой метод переопределяет метод базового класса.

Параметры

Параметры функций и конструкторов записываются в таком виде — сначала имя, потом тип параметра:

```
name: Type
```

В Kotlin, как и в Java, функции принимают нефиксированное количество параметров — varargs. Но обозначаются такие параметры немного иначе, чем в Java. В языке Kotlin надо использовать модификатор vararg. Внутри функции такие параметры выглядят как массив.

```
fun printStrings(vararg strings: String) {
    for(s in strings) {
        println(s)
    }
}
```

В Kotlin есть понятие «параметры по умолчанию». Это означает, что можно задать параметры метода или конструктора, которые будут подставлены компилятором, если мы не зададим их при вызове метода. Например, зададим город и температуру по умолчанию:

```
class Weather(val town: String = "Москва",
              val temperature: Int = 15)
```

Теперь можно создавать объект Weather, не передавая в конструктор параметры:

```
val w = Weather()
```

Или передавать только один. Второй параметр будет равен значению по умолчанию:

```
val w = Weather("Саратов")
```

Так создаются перегруженные методы, а не ряд методов с разным набором параметров. Можно сделать один и задать всем параметрам значения по умолчанию. В некоторых случаях классу требуется конструктор по умолчанию (без параметров). Вот как можно сделать это для погоды:

```
class Weather(
   val town: String = "",
   val temperature: Int = 0
)
```

Теперь создадим пустой класс:

```
val w = Weather()
```

Ещё одна полезная особенность — именованные аргументы. Имена параметров обычно указываются явно, при вызове функций. Это очень удобно, когда у функции большой список параметров, в том числе со значениями по умолчанию.

Рассмотрим следующую функцию:

```
fun setWeather(town: String,
               isHomeTown: Boolean = true,
               temperature: Int = 15) {
...
}
```

Мы можем вызвать её, используя аргументы по умолчанию:

```
setWeather("Москва")
```

Однако при вызове этой функции со всеми аргументами получится что-то вроде:

```
setWeather("Москва", false, 0)
```

Не совсем понятно.

Но можно записать вызов функции так:

```
setWeather("Москва", isHomeTown = false, temperature = 0)
```

Важно! *При вызове функции как с позиционными, так и с именованными аргументами, все позиционные аргументы должны располагаться перед первым именованным аргументом.*

Например, вызов f(1, y = 2) разрешён, а f(x = 1, 2) — нет.

Переменные и свойства

Объявление переменных в Kotlin не такое, как в Java. В начале объявления указывается ключевое слово var или val, затем — имя переменной, и далее — её тип или сразу значение:

```
val weatherList: List<Weather>
```

Если переменная инициализируется сразу, в месте объявления, тип можно не указывать. Компилятор выведет тип переменной из типа присвоенного значения:

```
var town = "Москва"
```

В этом случае тип переменной — String, потому что мы сразу присвоили ей значение типа String. В объявлении переменной типа не было, но он строго определён присвоенным значением. Присвоить ей значение другого типа не позволит компилятор:

```
town = 5 // ошибка компиляции
```

В Kotlin есть два типа переменных:

1. var — изменяемая, то есть значение ей присваивается бессчётное количество раз.
2. val — неизменяемая, final в терминах Java, значение ей присваивается только единожды.

Объект, на который она указывает, может меняться:

```
val weatherList = ArrayList<Weather>()
weatherList.add(Weather())
```

Создатели Kotlin рекомендуют использовать val, где это возможно. Такой подход позволяет избежать ошибок и написать более надёжный код.

Свойства классов

В Kotlin не используется понятие «переменная класса», но есть термин «свойство класса». Различие в том, что свойство содержит не только значение переменной, но и геттер с сеттером. Когда объявляем переменную типа var в классе, для неё автоматически создаются методы get и set. Мы их не видим, если не требуется их переопределять. Свойства класса могут объявляться прямо в конструкторе:

```
class MainViewState(val weatherList: List<Weather>)
```

Посмотрим, во что это компилируется:

```
public final class MainViewState {
  @NotNull
  private final List weatherList;

  @NotNull
  public final List getWeatherList() {
     return this.weatherList;
  }

  public MainViewState(@NotNull List weatherList) {
```

```
        this.weatherList = weatherList;
    }
}
```

Как видим, компилятор сгенерировал Java-класс. Есть приватное поле и getter для доступа к нему. Поле weatherList помечено модификатором final, так как в объявлении класса на Kotlin оно даётся с ключевым словом val — значит, доступно только для чтения, а инициализируется в конструкторе.

Важно! *В Android Studio можно посмотреть байт-код, куда компилируется код, написанный на Kotlin. Ещё есть декомпилятор, который может декомпилировать байт-код в код на Java.*

Для этого надо выбрать пункт меню Tools → Kotlin → Show Kotlin bytecode. Далее в открывшемся справа окне нажать Decompile.

Чтобы обратиться к свойству класса из кода на Kotlin, не надо вызывать getter явно. Обращаемся к нему, как к public-переменной в Java:

```
val viewState = ViewState(weatherList)
var someWeatherList = viewState.weatherList
```

Но «под капотом» вызывается метод getWeatherList(, который мы видели в декомпилированном классе.

Свойства классов должны проинициализироваться в момент создания класса: в конструкторе или прямым присвоением значения в месте объявления. Это сделано для безопасности, чтобы нельзя было обратиться к свойству, которое не проинициализировано. Но возможна отложенная инициализация через ключевое слово lateinit:

```
lateinit var viewModel: MainViewModel
```

Объявляя таким образом свойство, мы говорим компилятору, что берём на себя ответственность за инициализацию. Если обратиться к такому свойству до его инициализации, то выбросится UninitializedPropertyAccessException, и приложение «упадёт». Эту возможность надо использовать с осторожностью.

Методы get и set

Когда мы присваиваем значение переменной класса, часто требуется выполнить и конкретные действия. В Java делаем это в методах доступа к переменной. В Kotlin это выглядит иначе:

```
var weatherList: List<Weather> = listOf<Weather>()
    set(value) {
```

```
        field = value
        notifyDataSetChanged()
    }
```

Чтобы переопределить setter, используем ключевое слово set. Value в этом примере — значение, которое присваивается свойству weatherList, а field — это специальное поле, где хранится значение свойства weatherList.

Таким же образом переопределяется getter:

```
class ViewState(val weatherList: List<Weather>) {

    val hasData: Boolean
         get() = weatherList.size != 0
}
```

Здесь видим, что свойство hasData не имеет собственного значения и при обращении каждый раз вычисляет его, сравнивая массив с нулём.

Функции и переменные верхнего уровня

В Kotlin функции и переменные объявляются не только внутри классов, но и просто в файлах, которые на Kotlin имеют расширение .kt. Это означает, что можно создать файл example.kt со следующим содержанием:

```
package com.example.myapp

val applicationName = "MyApp"

fun getFullName(packageName: String, appName: String) {
    return packageName + appName
}
```

Теперь мы можем в любом месте приложения вызвать эту функцию или обратиться к такой переменной:

```
class SomeClass {

    fun doSomething() {
        val name = getFullName("com.example.", applicationName)
        ...

    }
}
```

Это полезная особенность языка: она позволяет заменить статические методы классов Utils, которые обычно есть в программах на языке Java, на более лаконичные функции.

Модификаторы видимости

В Kotlin есть четыре вида модификаторов доступа: public, internal, protected и private. Они применяются к классам, интерфейсам, конструкторам, свойствам и их сеттерам и функциям. Рассмотрим, как изменяют видимость модификаторы:

1. public виден везде.
2. internal виден внутри модуля. Это может быть модуль в Android-приложении или всё приложение, если оно состоит из одного модуля.
3. protected виден в классах-наследниках. Если в классе-наследнике не установлен другой модификатор, он остаётся protected.
4. private виден только внутри класса или файла, если это функция или переменная верхнего уровня.

Если модификатор не указывается, он будет public по умолчанию. Рассмотрим на примере:

```
open class Parent {
    private val first: String = "first"// видно только внутри класса Parent
    protected val second: String = "second"// видно внутри Parent и его
наследников
    val third: String = "third"// public по умолчанию — видно везде

    protected class Inner {// класс виден только наследникам Parent
        private val fourth: String = "fourth"// видно только внутри класса
Inner
    }
}
```

Модификатор inner используется для классов, объявленных внутри другого класса, и обозначает их как внутренние. По умолчанию классы, объявленные без модификатора inner, считаются самостоятельными — static в понятиях Java.

Data-классы

Часто мы применяем классы (POJO) как объект только для хранения данных. Чтобы использовать такой класс, надо создать геттеры и сеттеры для каждого поля класса, переопределить методы equals(), hashCode() и, желательно, toString(). Затем в классе получается больше бойлерплейт-кода,

чем кода, который мы действительно хотим использовать. Kotlin предлагает для таких случаев изящное решение — data class:

```
data class Note(val title: String, val note: String, val color: Int)
```

Ключевое слово data перед декларацией класса даёт компилятору понять, что мы хотим, чтобы он сгенерировал методы доступа к свойствам, а также методы equals() и hashCode(), toString(). equals() и hashCode() будут учитывать все свойства, объявленные в главном конструкторе.

Компилятор создаёт для data-классов ещё один метод — copy(), который добавляет копию этого объекта. Чтобы понять замысел создателей языка, вспомним, что hashCode() и equals() учитывают все свойства класса.

Представим ситуацию: есть HashMap, ключи в которой — это объекты. Если изменим какое-нибудь свойство такого объекта, больше не сможем получить значение из HashMap по нему, так как его hashCode изменился. Чтобы избежать таких ситуаций, создатели Kotlin рекомендуют использовать метод copy() для изменения объектов:

```
val newCopy = weather.copy(town = "Санкт-Петербург")
```

Здесь мы применили именованный параметр, чтобы изменить свойство town нашего класса Weather. Метод copy() использует по умолчанию в качестве параметров свойства копируемого объекта. Если не задать новых значений, получим точную копию объекта. Так, применяя именованные параметры, можно задать новые значения только для некоторых свойств класса, а остальные — просто не указывать. На выходе получаем объект со свойствами копируемого, кроме тех, что изменили.

Object

Это ключевое слово позволяет объявить класс и сразу же создать его экземпляр. Мы использовали его, чтобы создать синглтон:

```
object Repository
```

Посмотрим, во что это компилируется, если выбрать Tools → Kotlin → Show bytecode. Для упрощения опустим всю реализацию логики этого класса:

```
public final class Repository {

  public static final Repository INSTANCE;

  static {
```

```
        Repository var0 = new Repository();
        INSTANCE = var0;
    }
}
```

Как видим, это класс, экземпляр которого создаётся при загрузке класса. Это аналог статических классов в Java, так как в Kotlin нет статики в явном виде, как мы привыкли к этому в Java. Обратиться к такому объекту можно, как и в Java, просто по имени класса:

```
val repositoryData = Repository.getData()
```

Из кода на Java мы должны явно обратиться к экземпляру класса:

```
List<Weather> weatherData = Repository.INSTANCE.getData()
```

Companion object

В Kotlin нет ключевого слова static. Чтобы добавить члены класса, доступные без создания экземпляра класса, используется конструкция companion object:

```
class WeatherActivity : AppCompatActivity() {

    companion object {

        private val EXTRA_WEATHER = WeatherActivity::class.Java.name

        fun getIntent(context: Context, weather: Weather): Intent {
            val intent = Intent(context, WeatherActivity::class.Java)
            intent.putExtra(EXTRA_WEATHER, weather)
            return intent
        }
    }
}
```

Таким образом, внутри класса WeatherActivity создаётся объект с именем по умолчанию Companion. Можно задать другое название:

```
companion object Factory {
...
}
```

Получить доступ к членам companion object мы можем как к статическим членам в Java:

```
val intent = WeatherActivity.getIntent(this, weather)
```

Чтобы обратиться к companion object из Java, надо вызвать его прямо через имя. По умолчанию это Companion.

```
Intent intent = WeatherActivity.Companion.getIntent(this, weather)
```

Если мы задавали для companion object собственное имя, то вместо Companion надо использовать его. У companion object есть доступ ко всем приватным членам внешнего класса. Можем это использовать, когда требуется сделать инициализацию класса только через фабричные методы.

Объекты, объявленные через ключевое слово object, в том числе companion object, могут, как и другие классы, реализовывать интерфейсы и наследоваться от классов.

```
class Weather (val town: String, val temperature: Int) {

   companion object : Comparator<Weather> {
       override fun compare(o1: Weather, o2: Weather): Int {
           if (o1.temperature > o2.temperature) {
               return 1
           } else if (o1.temperature == o2.temperature) {
               return 0
           } else {
               return -1
           }
       }
   }
}
```

Анонимные классы

Через ключевое слово object создаются анонимные экземпляры классов:

```
private val textChangeListener = object : TextWatcher {
   override fun afterTextChanged(s: Editable?) {
       triggerSaveNote()
   }

   override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int,
after: Int) {
       // not used
   }

   override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count:
Int) {
       // not used
```

```
    }
}
```

Здесь мы создали анонимный класс, реализующий интерфейс TextWatcher, и присвоили его переменной. Таким же методом этот объект передаётся прямо в функцию:

```
titleEt.addTextChangedListener(object : TextWatcher {
    override fun afterTextChanged(s: Editable?) {
        triggerSaveNote()
    }
    override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int,
after: Int) {
        // not used
    }

    override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count:
Int) {
        // not used
    }
})
```

Созданный таким образом объект — не синглтон. При каждом вызове этого кода будет создаваться новый экземпляр такого объекта. Объекты, реализующие интерфейс, который содержит один метод, например, View.OnclickListener, не требуется создавать таким способом. Для них в Kotlin предусмотрены лямбда-выражения, которые мы рассмотрим позже.

If, then, when

В Kotlin такие конструкции, как if/then, try/catch и when, — это выражения, которые могут возвращать значение:

```
supportActionBar?.title = if (note != null) {
    SimpleDateFormat(DATE_TIME_FORMAT,
Locale.getDefault()).format(note.lastChanged)
} else {
    getString(R.string.new_note_title)
}
```

Здесь мы присваиваем значение, полученное из выражения if/else, переменной title. В Java такое невозможно.

В Kotlin, к сожалению, отсутствует тернарный оператор, поэтому вместо него используется выражение if/else. Когда всё выражение можно записать в одну строку, фигурные скобки опускаются:

```
val town = if (weather != null) weather.town else "Нет города"
```

Выражение when — замена конструкции switch в Java, но оно имеет более широкие возможности и может возвращать значение. Объявим enum. В Kotlin он мало чем отличается от своего аналога в Java, только необходимостью использовать слово class при объявлении:

```
enum class WeatherType {
   SUNNY,
   RAINY,
   CLOUDY,
   MISTY,
   SNOWY,
   HAILY
}
```

Дело в том, что в Kotlin есть мягкие ключевые слова, и enum — одно из них. Мы можем его использовать как обычное слово (например, имя переменной) в любом месте программы, а ключевым оно становится, только когда находится перед словом class. Как и в Java, enum в Kotlin могут содержать свойства и методы. Свойства объявляются при объявлении класса enum. Рассмотрим использование конструкции when:

```
val color = when(weather.type) {
   Color.SUNNY -> R.color.colorPrimary
   Color.RAINY -> R.color.color_violet
   Color.CLOUDY -> R.color.color_yellow
   Color.MISTY -> R.color.color_red
   Color.SNOWY -> R.color.color_pink
   Color.HAILY -> R.color.color_green
}
```

В отличие от Java, в конструкции when нет дополнительных слов case и break. В теле блока when пишем варианты, среди которых производится поиск. В качестве варианта может быть произвольное выражение, а не только константа, например, вызов функции. При совпадении выполняется только блок кода, который располагается за стрелкой этого варианта. When обычно вызывается без параметров, а в обеих частях вариантов могут быть выражения. В левой части размещается выражение, возвращающее boolean-значение, в правой — любое выражение, заключённое в фигурные скобки.

```
override fun onOptionsItemSelected(item: MenuItem): Boolean = when(item.itemId)
{
       android.R.id.home -> {
           onBackPressed()
           true
```

```
        }
        else -> super.onOptionsItemSelected(item)
    }
```

Если when используется как выражение, то есть мы ожидаем возвращаемое значение, то все ветви должны возвращать значение одного типа. Как и в других выражениях в Kotlin, например, лямбда-выражениях, возвращаемым значением выражения в фигурных скобках после стрелки будет последнее значение в этом выражении. В нашем случае — true. Слово return здесь опускается.

Если мы не можем описать все возможные варианты значений параметра, с которым вызван when, — в нашем случае — item.itemId, то в конце блока должна быть ветвь else. Эта ветвь выполнится, если ни один из вариантов выше не сработал.

В Kotlin также есть операторы in и is:

- in проверяет, входит ли элемент в коллекцию или интервал (об этом позже);
- is — принадлежит ли к определённому типу, по аналогии с instanceof в Java.

Конструкция when также используется с этими операторами:

```
when(item) {
    is Weather -> textView.setText(item.town)
    else -> textView.setText("Это не погода")
}
```

Циклы

В Kotlin нет такой реализации цикла for, как мы привыкли видеть в Java. Цикл for в Kotlin может итерироваться по любым последовательностям, которые имеют итератор. Такая последовательность должна иметь метод iterator(), он вернёт объект, у которого есть методы hasNext() и next(). Нам знакома такая реализация по конструкции foreach в Java.

```
val weatherList: List<Weather>
for(weather in weatherList) {
    print(weather.town)
}
```

Когда требуется организовать цикл из последовательности чисел, используется ClosedRange<Int> из стандартной библиотеки Kotlin. ClosedRange — это интерфейс, представляющий собой интервал элементов, реализующих интерфейс Comparable. Такая последовательность создаётся через оператор «..».

```
for(i in 1..10) {
    print("Hello Kotlin!")
}
```

Фраза Hello Kotlin! напечатается 10 раз. Чтобы организовать обратную последовательность, надо использовать оператор downTo. В обоих случаях можем указать шаг через оператор step:

```
for(i in 10 downTo 1 step 2) {
    print("Hello Kotlin!")
}
```

В этом случае Hello Kotlin! напечатается 5 раз. Операторы «..» и downTo добавляют последовательность, включающую числа, указанные при создании. Часто требуется обозначать конец последовательности, не включая его, например, при итерации по коллекции. Для таких случаев в Kotlin предусмотрен оператор until:

```
for (i in 0 until weatherList.size) {
  if (weatherList[i] == weather) {
      weatherList.set(i, weather)
      return
  }
}
```

Цикл while в Kotlin работает так же, как и в Java, и здесь мы его рассматривать не будем.

Практическое задание

Прежде чем приступать к первому практическому заданию, надо определиться с курсовым проектом. Если всё ещё чувствуете себя неуверенно в коде, то можете пойти по более лёгкому пути и писать своё приложение «Погода» на основе примеров из урока. Или взяться за совершенно новый проект «Поиск по фильмам», для которого даются практические задания. Последний проект потребует от вас большей самостоятельности и вовлечённости.

Что это будет?

1. Погодное приложение, как в примерах на занятиях, но с вашими улучшениями или дополнениями.
2. Приложение для поиска фильмов, по которому вам будут даваться все практические задания.

Выбор за вами!

1. Создать новый проект в Android Studio без поддержки Kotlin.

2. Сконфигурировать Kotlin в новом проекте (лучше вручную).
3. Перевести MainActivity на Kotlin.
4. Добавить кнопку в разметку и повесить на неё clickListener в Activity.
5. Потренироваться в создании классов и функций, описанных в уроке, и убедиться, что всё работает. Например, создать тестовое приложение:
    a. Сформировать data class с двумя свойствами и вывести их на экран приложения.
    b. Создать Object. В Object вызвать copy и вывести значения скопированного класса на экран.
    c. Вывести значения из разных циклов в консоль, используя примеры из методических материалов.
6. Изучить API погоды «Яндекса», посмотреть примеры и зарегистрироваться в качестве разработчика, получить свой ключ разработчика.
7. Изучить API AMDB и зарегистрироваться в качестве разработчика, подключиться к API.
8. Определиться с экранами и инструментарием своего будущего приложения с фильмами на основе возможностей API.

Задача для дополнительного обучения

Переведите проект с заметками на Kotlin. Курс «Базовый уровень».

Дополнительные материалы

1. Официальная документация Google.
2. Дмитрий Жемеров, Светлана Исакова «Kotlin в действии».
3. Упражнения по Kotlin Koans.
4. Статья The one and only object.

Используемые источники

1. Develop Android apps with Kotlin.
2. Дмитрий Жемеров, Светлана Исакова «Kotlin в действии».