GeekBrains

Android на Kotlin

Типы данных, коллекции, null safety, дженерики и интерфейсы

На этом уроке

1. Изучим типы данных в Kotlin.
2. Познакомимся с концепциями безопасности, интерфейсами и дженериками в Kotlin.
3. Рассмотрим массивы и коллекции.

Оглавление

Типы данных в Kotlin

Null safety

Создавая Kotlin, разработчики старались сделать его максимально безопасным, поэтому в языке есть понятие null safety. Безопасность достигается посредством разделения типов переменных на поддерживающие хранение null и не поддерживающие. Если мы напишем такой код, то получим ошибку компиляции:

```
var notNullable: String = ""
notNullable = null  // Ошибка компиляции
```

Чтобы в переменной хранить null, надо объявить её соответствующим образом:

```
var nullable: String? = " "
```

Если попытаться присвоить переменной, не поддерживающей null, значение переменной с поддержкой nullable-типов, компилятор не даст скомпилировать такой код:

```
nontNullable = nullable // Ошибка компиляции
```

У нас также не получится вызвать методы объекта, хранящегося в такой переменной:

```
val length = nullable.length // Ошибка компиляции
```

Чтобы присвоить значение nullable-переменной, не поддерживающей null, надо сначала проверить, не содержит ли nullable-переменная null:

```
if (nullable != null) {
    length = nullable.length // OK
    nonNullable = nullable   // OK
}
```

После этого в области видимости проверки переменная nullable будет рассматриваться компилятором как тип, не поддерживающий null. Можно безопасно обращаться к хранимому в переменной объекту.

Защита от NullPointerException реализуется на уровне компилятора. Это касается не только переменных, но и выражений и вызовов функций. Компилятор не даст передать в функцию значение переменной, которая поддерживает nullable-типы, если в определении функции заявлены только nonNull-параметры. Он также не позволит присвоить переменной, не поддерживающей null, значение, полученное из выражения, которое может возвращать null.

```
var name: String = "Иван"
var fullName: String? = ""
fun checkStirng(s: String): String? {
    ...
}

checkString(fullName)  // Ошибка компиляции
name = checkString(name) // Ошибка компиляции
```

Это очень удобно. В Java нет поддержки null-безопасности на уровне языка, поэтому там приходится использовать аннотации, что не только добавляет лишние строки кода, но и не гарантирует безопасности: код прекрасно собирается и принимает null даже там, где его не должно быть. Аннотация только подсвечивает проблемные места. В Kotlin же такой код просто не соберётся.

На первый взгляд, не очень удобно каждый раз писать проверку на null: if (s != null). И Kotlin предоставляет более удобные инструменты для работы с nullable-типами.

Оператор безопасного вызова

Первый такой инструмент — оператор безопасного вызова. Выглядит как вопросительный знак с последующей точкой:

```
val name: String? = "John"
val nameLength: Int? = name?.length
```

Производит проверку на null перед вызовом метода. Если значение переменной равно null, то вместо вызова исключения это выражение просто вернёт null. То есть если name == null, то nameLength будет == null или == 4. Очень удобная и быстрая проверка, которая часто используется в коде. В своём приложении мы тоже воспользуемся этим оператором.

Оператор «Элвис»

Ещё один удобный оператор для работы с null safety. «Элвис» записывается знаком вопроса с последующим двоеточием — ?:. Если задействовать воображение и посмотреть на запись под определённым углом, можно найти сходство с образом Пресли:

```
val nameLength: Int = name?.length ?: 1
```

Принцип работы следующий:

- если выражение слева вернёт не null, то мы получим длину имени;
- если же null, то вернётся значение выражения справа от оператора.

Таким образом, легко реализуется проверка на null и возврат значения по умолчанию. Исходя из примера выше, если name == null, то nameLength будет == 1 или == 4.

Оператор безопасного приведения типов

Для приведения типов в Kotlin используется оператор as. Но если приведение выполнить нельзя, будет брошено исключение ClassCastException. Для таких случаев используется оператор as?, который приводит типы, если это возможно, иначе, возвращает null:

```
val intentData: Note? = intent.getSerializable(EXTRA_NOTE) as? Note
```

Если в объекте Intent не будет передан параметр, то вместо выбрасывания ClassCastException переменной intentData просто присвоится значение null.

Утверждение «!!»

Для случаев, когда значение переменной точно не равно null, есть оператор «!!». Он позволяет сказать компилятору, что эта переменная не содержит null:

```
length = nullable!!.length
nonNullable = nullable!!
```

В обоих случаях код компилируется без ошибок. Но нет никаких гарантий, что исключение не появится во время выполнения.

Двумя восклицательными знаками в этом утверждении создатели Kotlin предупреждают, что использовать его опасно. Лучше избегать, даже если есть 100% уверенность, что там нет значения null.

Null safety при работе с кодом на Java

Когда мы работаем с кодом на Java, компилятор старается руководствоваться аннотациями типа @Nullable или @NonNull, которые есть в Java-коде. Но эти аннотации есть далеко не везде. Если их нет, компилятор не может определить, поддерживает ли переменная, объявленная в Java-коде, значение null. В таких случаях компилятор считает переменную платформенным типом и оставляет за разработчиком решение, как с ней работать. Он не будет выдавать ошибок компиляции при работе с платформенными типами. Ответственность за риск получить NullPointerException — полностью на разработчике.

Базовые типы

Базовые типы integer, float, double, character и другие в Kotlin не подразделяются на примитивы и обёртки для них. Вместо этого в языке есть типы Int, Char и остальные — с такими же названиями, как у классов-обёрток для примитивов в Java. Они всегда ведут себя как классы, но это не означает, что в Kotlin вместо примитивов всегда используются классы-обёртки. **Компилятор Kotlin использует примитив для представления значения, где это возможно.** Но примитивы в Java не хранят null — только ссылка. Если в Kotlin мы объявляем базовый тип с поддержкой null, то есть знаком «?» после указания типа, такая переменная всегда будет представлена классом-обёрткой.

В целом базовые типы в Kotlin работают так же, как в Java, но есть и отличия. В Kotlin нет автоматического приведения типов. Если требуется присвоить значение типа Int переменной Double, надо сделать явное приведение типа:

```
val integer = 1
val double: Double = integer.toDouble()
```

У базовых типов есть много функций, в том числе для преобразования в другие базовые типы.

Типы Any, Unit

Any — родительский тип для всех типов в Kotlin. Так как в этом языке нет различия между примитивными типами и ссылочными, от Any унаследованы и базовые типы. Если решите присвоить переменной с типом Any значение базового типа, например, Int, будет использована обёртка над примитивом.

Тип Any — аналог Object в Java, но имеет только три метода: toString(), equals() и hashCode(). Если потребуется использовать остальные методы из класса Object, то придётся явно привести тип к Object.

Тип Unit используется, когда надо указать, что функция не возвращает ничего полезного. Это аналог void в Java, а Unit — объект, который всегда один на всё приложение. Если тело функции не возвращает значение, и не указывается инструкция return, то компилятор автоматически вернёт из функции Unit.

Работа с коллекциями

Массивы

Массивы в Kotlin представлены классом Array. Создать массив можно посредством функции из стандартной библиотеки arrayOf(). Доступ к элементам массива осуществляется через методы get() и set(), которые, как и в других коллекциях, представляются путём использования [ ]. Размер массива показывает свойство size:

```
val phrase: Array<String> = arrayOf("I", "love", "Kotlin")
val lang = phrase[2]
phrase[1] = "know"
val wordCount = phrase.size
```

Коллекции

Коллекции в Kotlin представлены реализациями коллекций из Java, но работают они через интерфейсы, определённые в Kotlin. Основное отличие от Java-коллекций: в Kotlin они разделены на изменяемые и неизменяемые. Неизменяемые коллекции реализуют интерфейс kotlin.collections.Collection, который не имеет методов для изменения: add(), set() и т. д. Изменяемые коллекции реализуют интерфейс kotlin.collections.MutableCollection, где есть все методы коллекций Java.

Класс Map в Kotlin тоже может быть в двух вариантах: Map (неизменяемый) и MutableMap (изменяемый).

В интересах безопасности авторы Kotlin рекомендуют использовать изменяемые коллекции только в случае, если планируется вносить в них корректировки.

Но так как «под капотом» используются коллекции Java, которые не разделяются на изменяемые и неизменяемые, остаётся вероятность, что где-то в другом месте ссылка на коллекцию будет

объявлена изменяемой, и коллекция всё-таки изменится. Неизменяемость коллекции также не гарантирует такой же статус её элементов:

```
class Person(val name: String, var age: Int)
val people: List<Person> = listOf(Person("Василий", 25), Person("Татьяна", 23))
people[0].age = 26
```

В этом примере коллекция содержит те же элементы, но их свойства изменились.

Коллекция создаётся через конструктор:

```
val list: List<String> = ArrayList()
val mutableSet: MutableSet<Int> = HashSet()
```

Или посредством одной из специальных функций:

Тип	Неизменяемая	Изменяемая
List	listOf()	mutableListOf(), arrayListOf()
Set	setOf()	mutableSetOf(), hashSetOf(), linkedSetOf(), sortedSetOf()
Map	mapOf()	mutableMapOf(), hashMapOf(), linkedMapOf(), sortedMapOf()

Все методы в аргументах принимают список элементов для добавления через запятую. А методы, создающие Map, принимают в качестве параметров экземпляры класса Pair, который содержит ключ и значение соответственно. О Pair и Triple поговорим на другом занятии.

Функции-расширения для работы с коллекциями

В стандартной библиотеке Kotlin есть функции-расширения для интерфейсов коллекций, которые позволяют обрабатывать их в функциональном стиле. Наиболее часто используются filter и map. Рассмотрим сигнатуры этих функций:

```
public inline fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T>
```

Практически все они принимают в качестве параметров функции для действий над коллекциями. Функция-расширение проходит по коллекции и вызывает переданную функцию, передав ей очередной элемент коллекции.

Filter принимает функцию-предикат, которая возвращает true или false. Если предикат вернул true, то элемент добавляется в итоговую коллекцию. Пройдя по коллекции, функция возвращает список,

состоящий из элементов, для которых предикат вернул true. Например, можем отобрать все заметки с непустым телом:

```
val noEmptyBodies = persons.filter { it.person.isNotEmpty() }
```

Так как элемент коллекции, переданный в предикат, — единственный аргумент, обратимся к нему по имени it внутри функции-предиката. Функция String.isNotEmpty() — тоже функция-расширение из стандартной библиотеки.

Функция map позволяет трансформировать элементы коллекции посредством переданной функции:

```
public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R>
```

Результат работы этой функции — список, содержащий элементы, возвращённые переданной функцией. Например, можно преобразовать список документов, полученных с сервера, в список заметок:

```
val notes = documents.map { it.toObject(Note::class.java) }
```

Этот код пройдёт по коллекции документов, вызовет у каждого элемента функцию toObject() и вернёт коллекцию, содержащую элементы класса Note.

Функции-расширения вызываются последовательно. Например, чтобы найти все заметки с непустым телом, не создавая промежуточной переменной:

```
val noEmptyBodies = documents.map { it.toObject(Note::class.java) }
                             .filter { it.note.isNotEmpty() }
```

Есть другой тип функций, возвращающие не коллекцию, а значение. Это find, count, any и all. Функция find вернёт первый элемент, который соответствует переданному предикату:

```
val kotlinNote: Note = notes.find { it.note.contains(" Kotlin ") }
```

Эта функция найдёт первую заметку, тело которой содержит слово Kotlin.

Функция count вернёт количество элементов, удовлетворяющих условию предиката:

```
val kotlinMention: Int = notes.count { it.note.contains(" Kotlin ") }
```

Функции any и all вернут true, если один или все элементы соответствуют условиям предиката:

```
val isKotlinMentioned = notes.any { it.note.contains(" Kotlin ") }
val isAllAboutKotlin = notes.all { it.note.contains(" Kotlin ") }
```

Мы рассмотрели только часть функций для работы с коллекциями, чтобы разобрать их принципы. Ознакомиться с коллекциями и функциями-расширениями для них можно в официальной документации.

Простое форматирование строк

Для простой работы со строками в Kotlin есть строковые шаблоны, которые позволяют использовать переменные в строковых литералах:

```
override fun saveNote(note: Note): {
    Log.d(TAG, "Note $note is saved")
}
```

В строковый литерал добавилась переменная note, обозначенная знаком $. Эта запись равносильна такой:

```
Log.d(TAG, "Note "  +  note + "  is saved")
```

Ещё в строковых литералах используются целые выражения. Для этого их требуется заключать в фигурные скобки:

```
private val TAG = "${MainActivity::class.java.simpleName} :"
```

В строку добавится имя класса. Если понадобится использовать в строковом литерале знак $, его надо экранировать: \$.

Строковые значения из предыдущего урока записываются и таким образом:

```
cityCoordinates.text = String.format(
   getString(R.string.city_coordinates),
   weatherData.city.lat.toString(),
   weatherData.city.lon.toString()
)
```

```
cityCoordinates.text = "$weatherData.city.lat $weatherData.city.lon"
```

Передавать значения в Textview через конкатенацию в этом случае не рекомендуется. Код показан для сравнения и примера.

Интерфейсы

Интерфейсы в Kotlin включают в себя функции — абстрактные и с реализацией, а также свойства, которые должны быть абстрактными или иметь методы доступа. Абстрактные функции и методы переопределяются в классах, реализующих интерфейс:

```
interface FlyingVehicle {

    val engine: Engine

    val greeting: String
        get() = "Hello from the air!"

    fun takeOff(): Unit
    fun land(): Unit
    fun getHeight(): Double

    fun warmUp() {
        engine.start()
    }
}
```

Реализуемый интерфейс указывается, как и наследуемый класс, через двоеточие после названия класса. Если класс реализует несколько интерфейсов, они перечисляются через запятую. Переопределяемые свойства и методы обязательно обозначаются ключевым словом override:

```
class Jet : FlyingVehicle, Cargo {
    override val engine: Engine = JetEngine()

    override fun takeOff() {
        /.../
    }

    override fun land() {
        /.../
    }

    override fun getHeight() {
        /.../
```

```
    }
}
```

Generics

В Kotlin, как и в Java, используются классы с generic-типами, и инструментарий у них похожий:

```
open class BaseViewState<T>(val data: T, val error: Throwable?)
```

Экземпляр такого класса создаётся следующим образом:

```
val viewState = BaseViewState<String>("Hi!", null)
```

Так как generic-тип выводится из переданного параметра, его можно опустить:

```
val viewState = BaseViewState("Hi!", null)
```

В Kotlin также задаётся верхняя граница принимаемого generic-типа:

```
open class BaseViewState<T : CharSequence>(val data: T, val error: Throwable?)
```

Мы объявили, что этот класс принимает только наследников CharSequence в качестве generic-типа.

Generic-типы также используются в функциях:

```
fun <T> doSomething(item: T): List<T> {
...
}
```

Вариативность в дженериках

При использовании generic-типов логично полагать, что List<String> — подтип List<Object>. Но сделать, как в примере ниже, Java не позволит — это небезопасно:

```
List<String> strings = new ArrayList();
List<Object> objects = strings;  // Ошибка компиляции
```

```
objects.add(8);
String s = strings.get(0);      // ClassCastException
```

Generic-типы в Java инвариантны — это не подтипы друг друга. Для таких случаев в Java есть маски:

```
List<String> strings = new ArrayList();
List<? extends Object> objects = strings; // Ok
objects.add(8);                           // Ошибка
Object obj = objects.get(0);              // Ok
```

Этот код скомпилируется без ошибок, но мы сможем читать только объекты из objects. Поэтому и безопасно: у нас получится использовать подклассы Object в качестве Object. Такие типы ковариантны друг другу. В Java есть и другая маска — для ограничения нижнего предела:

```
List<CharSequence> chars = new ArrayList<>();
List<? super String> strings = chars;
strings.add("Hi!");           // OK
String st = strings.get(0);   // Ошибка
```

В этом случае, наоборот: есть возможность безопасно добавить объект типа String, так как мы знаем, что в переменной хранится коллекция супертипов по отношению к String, и всегда можем использовать String в качестве CharSequence. Такое отношение типов называется контравариантностью.

В Java это работает только в месте использования, в остальных случаях List<Object> и List<String> всегда инвариантны.

В Kotlin есть более обширные возможности:

```
class Producer<T> {
  fun produce(): T {
   /.../
  }
}
```

Этот класс только производит объекты типа T. Посмотрим, можно ли сделать так, чтобы Producer<String> вернул экземпляр класса String, который представляет собой подтип Any:

```
val anyProducer: Producer<Any> = Producer<String>()
val obj: Any = anyProducer.produce()
```

В Kotlin это запрещено по тем же причинам, что и в Java. Но в Kotlin есть понятие вариативности в месте объявления. Мы можем сделать так:

```
class Producer<out T> {
  fun produce(): T {
      /.../
  }
}
```

И тогда код, приведённый выше, станет валидным. Объявив параметр как out T, мы указали, что тип T используется только в качестве возвращаемого значения в классе Producer. Следовательно, Producer<String> — это подкласс Producer<Any>, потому что мы можем безопасно получать объекты типа String и использовать их в качестве Any. Такой класс называется ковариантным в параметре T.

Можно объявить параметр, который принимается только членами класса:

```
class Consumer<in T> {
    fun consume(param: T) {
        /.../
    }
}
```

Параметр In T обозначает, что в этом классе тип T используется только в качестве принимаемого параметра. Следовательно, Consumer<Any> используется там же, где и Consumer<String>. Consumer<Any> — подкласс Consumer<String>. Такой класс называется контравариантным в параметре T.

В Kotlin есть вариативность в месте использования, как и в Java:

```
fun consumeList(list: List<out CharSequence>) {
    /.../
}
```

Объявленная таким образом функция способна принимать List с любыми наследниками CharSequence, но внутри функции list доступен только для чтения. Это аналог записи в Java List<? Extends CharSequence>.

Аналогично:

```
fun produce(list: MutableList<in String>) {
    /.../
}
```

Эта функция примет любой List, параметризованный супертипом по отношению к String, так как в такой List всегда можно положить String. Такая запись аналогична List<? Super String> в Java.

Star projection

Если тип параметра неизвестен в момент исполнения, используется проекция со звёздочкой: List<*>. Тип LIst<*> обозначает List, содержащий элементы любого типа. Но так как этот объект содержит элементы конкретного типа, обозначенные в момент создания, это накладывает ограничения на его использование.

Мы не знаем, объекты какого типа содержатся в этом списке, и не можем добавить туда новые объекты: их тип может не совпасть с уже имеющимся. У нас есть возможность только читать объекты из этого списка. А возвращаемый тип будет Any?, потому что он считается родительским для всех типов. Знак вопроса говорит, что объект, скорее всего, == null. При неизвестном типе безопаснее рассматривать его только так:

```
when(t) {
    is Success<*> ->
        ...
    is Error ->
        ...
}
```

Практика

Добавим новый инструментарий в наше приложение. Теперь в приложении будет два экрана:

- со списком городов;
- с погодой выбранного города.

Сначала изменим модель, так как это ядро нашего приложения. От него изменения пойдут в более высокие слои. Добавим два метода:

```
package com.example.androidwithkotlin.model

data class Weather(
    val city: City = getDefaultCity(),
    val temperature: Int = 0,
    val feelsLike: Int = 0
)

fun getDefaultCity() = City("Москва", 55.755826, 37.617299900000035)
```

```
fun getWorldCities(): List<Weather> {
    return listOf(
        Weather(City("Лондон", 51.5085300, -0.1257400), 1, 2),
        Weather(City("Токио", 35.6895000, 139.6917100), 3, 4),
        Weather(City("Париж", 48.8534100, 2.3488000), 5, 6),
        Weather(City("Берлин", 52.52000659999999, 13.404953999999975), 7, 8),
        Weather(City("Рим", 41.9027835, 12.496365500000024), 9, 10),
        Weather(City("Минск", 53.90453979999999, 27.561524400000053), 11, 12),
        Weather(City("Стамбул", 41.0082376, 28.97835889999999), 13, 14),
        Weather(City("Вашингтон", 38.9071923, -77.03687070000001), 15, 16),
        Weather(City("Киев", 50.4501, 30.523400000000038), 17, 18),
        Weather(City("Пекин", 39.90419989999999, 116.40739630000007), 19, 20)
    )
}

fun getRussianCities(): List<Weather> {
    return listOf(
        Weather(City("Москва", 55.755826, 37.617299900000035), 1, 2),
        Weather(City("Санкт-Петербург", 59.9342802, 30.335098600000038), 3, 3),
        Weather(City("Новосибирск", 55.00835259999999, 82.93573270000002), 5, 6),
        Weather(City("Екатеринбург", 56.83892609999999, 60.60570250000001), 7,
8),
        Weather(City("Нижний Новгород", 56.2965039, 43.936059), 9, 10),
        Weather(City("Казань", 55.8304307, 49.06608060000008), 11, 12),
        Weather(City("Челябинск", 55.1644419, 61.4368432), 13, 14),
        Weather(City("Омск", 54.9884804, 73.32423610000001), 15, 16),
        Weather(City("Ростов-на-Дону", 47.2357137, 39.701505), 17, 18),
        Weather(City("Уфа", 54.7387621, 55.972055400000045), 19, 20)
    )
}
```

Это методы, которые возвращают массивы городов: русских или зарубежных.

Обратите внимание, как мы создаём список: listOf. Создание аналога ArrayList в Kotlin с возможностью читать только данные списка, но не добавлять в него новые. Это так называемый immutable (неизменяемый список). Если понадобится создать изменяемый список, используем стандартный метод mutableListOf. То же самое относится и к другим коллекциям: setOf/MutableSetOf, mapOf/mutableMapOf. В Kotlin можно применять и стандартные коллекции Java. Например, для создания ArrayList<> достаточно написать:

```
fun getWorldCities(): ArrayList<Weather> {
    return arrayListOf(
        Weather(City("Лондон", 51.5085300, -0.1257400), 0, 0),
        Weather(City("Токио", 35.6895000, 139.6917100), 0, 0),
        Weather(City("Париж", 48.8534100, 2.3488000), 0, 0),
        Weather(City("Берлин", 52.52000659999999, 13.404953999999975), 0, 0),
        Weather(City("Рим", 41.9027835, 12.496365500000024), 0, 0),
        Weather(City("Минск", 53.90453979999999, 27.561524400000053), 0, 0),
        Weather(City("Стамбул", 41.0082376, 28.97835889999999), 0, 0),
```

```
        Weather(City("Вашингтон", 38.9071923, -77.03687070000001), 0, 0),
        Weather(City("Киев", 50.4501, 30.523400000000038), 0, 0),
        Weather(City("Пекин", 39.90419989999999, 116.40739630000007), 0, 0)
    )
}
```

Мы добавили эти методы не в класс Weather, а просто снаружи класса. Как говорилось на прошлом занятии, в Kotlin нет чёткого разделения на файлы и классы. Можно писать код так, как удобно. До того, как мы определили эти методы, Weather отображался как класс, теперь в структуре проекта он указывается как файл. Но принципиальной разницы нет — мы просто добавили методы туда, где они уместнее.

Теперь очередь репозитория и его имплементации. Изменим тип возвращаемых данных и разделим получение локальных данных на два метода по типу городов, которые хотим отображать:

```
interface Repository {
    fun getWeatherFromServer(): Weather
    fun getWeatherFromLocalStorageRus(): List<Weather>
    fun getWeatherFromLocalStorageWorld(): List<Weather>
}

class RepositoryImpl : Repository {

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

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

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

Изменим тип данных для состояния приложения:

```
sealed class AppState {
    data class Success(val weatherData: List<Weather>) : AppState()
    data class Error(val error: Throwable) : AppState()
    object Loading : AppState()
}
```

Доработаем ViewModel:

```
class MainViewModel(
    private val liveDataToObserve: MutableLiveData<AppState> = MutableLiveData(),
    private val repositoryImpl: Repository = RepositoryImpl()
) :
    ViewModel() {

    fun getLiveData() = liveDataToObserve

    fun getWeatherFromLocalSourceRus() = getDataFromLocalSource(isRussian = true)

    fun getWeatherFromLocalSourceWorld() = getDataFromLocalSource(isRussian =
false)

    fun getWeatherFromRemoteSource() = getDataFromLocalSource(isRussian = true)

    private fun getDataFromLocalSource(isRussian: Boolean) {
        liveDataToObserve.value = AppState.Loading
        Thread {
            sleep(1000)
            liveDataToObserve.postValue(AppState.Success(if (isRussian)
repositoryImpl.getWeatherFromLocalStorageRus() else
repositoryImpl.getWeatherFromLocalStorageWorld()))
        }.start()
    }
}
```

Мы используем именованные аргументы (isRussian), чтобы увидеть, какой булинь мы передаём в качестве аргумента. А в самом методе getDataFromLocalSource применяем if как возвращаемое выражение, что в Java невозможно.

В новом потоке вызываем sleep(1000): останавливаем поток на секунду, чтобы имитировать процесс загрузки в приложении.

Теперь переходим к UI. Переименуем main_fragment в fragment_details.xml, а MainFragment — в DetailsFragment. Теперь это будет экран с деталями погоды. К этому экрану мы вернёмся позже. А пока закомментируем в нём весь код. Потому что для начала нам требуется создать список городов на главном экране. Для этого добавим макет fragment_main.xml. В проекте есть иконки для FAB в виде векторных рисунков, но можно добавить собственные:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.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">

    <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" />

    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/mainFragmentFAB"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/fab_margin"
        android:src="@drawable/ic_russia"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:maxImageSize="@dimen/fab_icon_size" />

    <FrameLayout
        android:id="@+id/mainFragmentLoadingLayout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@color/loadingBackground"
        android:visibility="gone"
        tools:visibility="visible">

        <ProgressBar
            style="?android:attr/progressBarStyleLarge"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center" />
    </FrameLayout>
</androidx.constraintlayout.widget.ConstraintLayout>
```

Добавим и элемент списка fragment_main_recycler_item.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/itemContainer"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <androidx.appcompat.widget.AppCompatTextView
        android:id="@+id/mainFragmentRecyclerItemTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="@dimen/recycler_item_padding"
        android:textSize="@dimen/recycler_item_text_size"
        app:layout_constraintStart_toStartOf="parent"
```

```
        app:layout_constraintTop_toTopOf="parent"
        tools:text="Москва" />

    <View
        android:layout_width="match_parent"
        android:layout_height="@dimen/recycler_item_line_height"
        android:layout_marginEnd="@dimen/recycler_item_line_margin_end"
        android:background="@android:color/darker_gray"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"

app:layout_constraintTop_toBottomOf="@+id/mainFragmentRecyclerItemTextView" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Ресурсы:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="city_name_margin_top">50dp</dimen>
    <dimen name="margin_15_dp">15dp</dimen>
    <dimen name="temperature_value_text_size">40sp</dimen>
    <dimen name="feels_like_text_size">30sp</dimen>
    <dimen name="fab_icon_size">56dp</dimen>
    <dimen name="fab_margin">15dp</dimen>
    <dimen name="recycler_item_padding">10dp</dimen>
    <dimen name="recycler_item_text_size">25sp</dimen>
    <dimen name="recycler_item_line_height">1dp</dimen>
    <dimen name="recycler_item_line_margin_end">20dp</dimen>
</resources>
```

Теперь создадим адаптер. В нём для нас нет ничего нового. Метод передачи массива с данными для отображения и обработка нажатия на элемент списка:

```
class MainFragmentAdapter :
    RecyclerView.Adapter<MainFragmentAdapter.MainViewHolder>() {

    private var weatherData: List<Weather> = listOf()

    fun setWeather(data: List<Weather>) {
        weatherData = data
        notifyDataSetChanged()
    }

    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): MainViewHolder {
        return MainViewHolder(
            LayoutInflater.from(parent.context)
```

```
                .inflate(R.layout.fragment_main_recycler_item, parent, false) as
View
        )
    }

    override fun onBindViewHolder(holder: MainViewHolder, position: Int) {
        holder.bind(weatherData[position])
    }

    override fun getItemCount(): Int {
        return weatherData.size
    }

    inner class MainViewHolder(view: View) : RecyclerView.ViewHolder(view) {

        fun bind(weather: Weather) {

itemView.findViewById<TextView>(R.id.mainFragmentRecyclerItemTextView).text =
weather.city.city
            itemView.setOnClickListener {
                Toast.makeText(
                    itemView.context,
                    weather.city.city,
                    Toast.LENGTH_LONG
                ).show()
            }
        }
    }
}
```

Обратите внимание, как мы обращаемся к элементу списка в массиве `weatherData` метода `onBindViewHolder`: **не вызываем методы `get` или `set` напрямую, а пользуемся синтаксическим «сахаром» в виде квадратных скобок**.

Создадим новый фрагмент MainFragment. Наш главный фрагмент теперь выглядит так. Добавилось два свойства:

- `adapter`, который мы сразу инициализируем;
- флаг `isDataSetRus`, который следит за «подгрузкой» тех или иных данных.

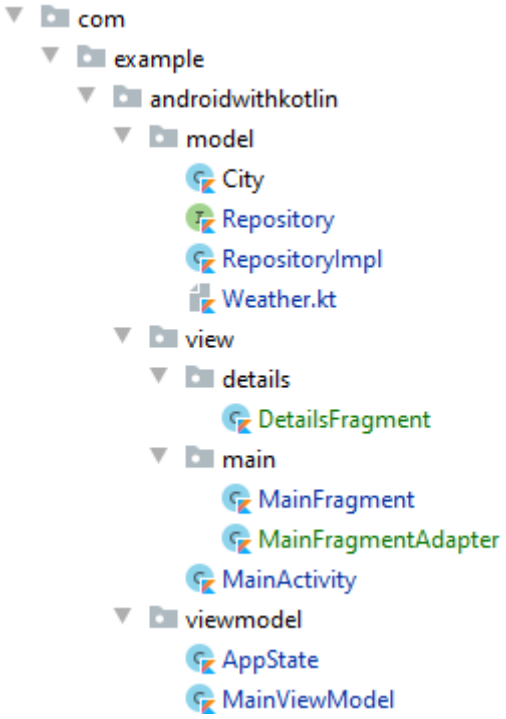
Весь основной код мы перенесли в callback `onViewCreated`. То есть, когда фрагмент будет готов к отображению, присвоим адаптер нашему списку, повесим листенер на FAB и запросим данные у ViewModel. Остальные методы говорят сами за себя:

```
class MainFragment : Fragment() {

    private var _binding: FragmentMainBinding? = null
    private val binding get() = _binding!!
```

```
    private lateinit var viewModel: MainViewModel
    private val adapter = MainFragmentAdapter()
    private var isDataSetRus: Boolean = true

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding = FragmentMainBinding.inflate(inflater, container, false)
        return binding.getRoot()
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        binding.mainFragmentRecyclerView.adapter = adapter
        binding.mainFragmentFAB.setOnClickListener { changeWeatherDataSet() }
        viewModel = ViewModelProvider(this).get(MainViewModel::class.java)
        viewModel.getLiveData().observe(viewLifecycleOwner, Observer {
renderData(it) })
        viewModel.getWeatherFromLocalSourceRus()
    }

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

    private fun renderData(appState: AppState) {
        when (appState) {
            is AppState.Success -> {
                binding.mainFragmentLoadingLayout.visibility = View.GONE
                adapter.setWeather(appState.weatherData)
            }
            is AppState.Loading -> {
                binding.mainFragmentLoadingLayout.visibility = View.VISIBLE
            }
            is AppState.Error -> {
                binding.mainFragmentLoadingLayout.visibility = View.GONE
                Snackbar
                    .make(binding.mainFragmentFAB, getString(R.string.error),
Snackbar.LENGTH_INDEFINITE)
                    .setAction(getString(R.string.reload)) {
viewModel.getWeatherFromLocalSourceRus() }
                    .show()
            }
        }
    }
```

```
    companion object {
        fun newInstance() =
            MainFragment()
    }
}
```

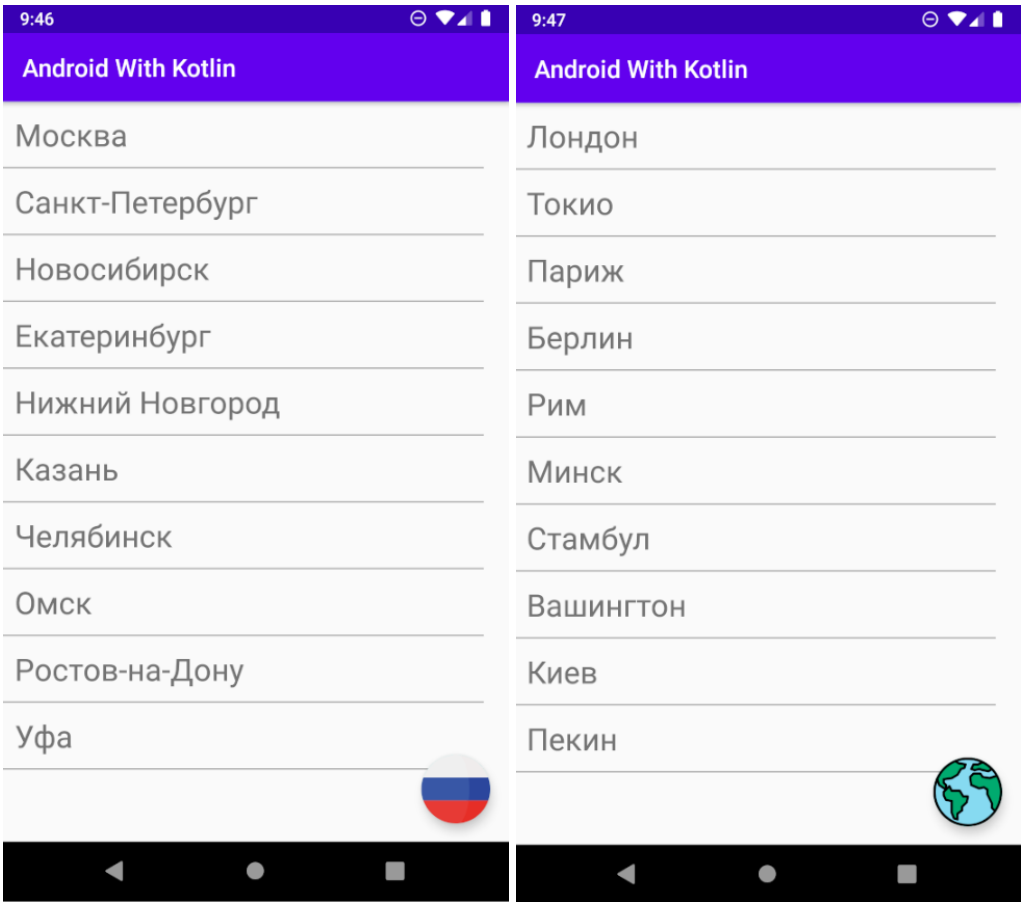
Немного перераспределили файлы по пакеджам:



В MainActivity проверяем, что запускается именно MainFragment:

```
.replace(R.id.container, MainFragment.newInstance())
```

Запускаем и проверяем, как это работает:



Теперь настало время второго экрана. Это наш изначальный фрагмент, доработанный до подходящего инструментария: по нажатию на элемент списка мы будем открывать новое окно и передавать туда данные для отображения города и погоды в нём. Начнём с макетов и ресурсов. Добавим непрозрачный фон в наш фрагмент, чтобы отображать его поверх MainFragment:

```
<FrameLayout xmlns:android="http://schemas.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"
   android:background="@color/fragmentBackground">
...
```

Ресурсы:

```
<resources>
  <color name="colorPrimary">#6200EE</color>
  <color name="colorPrimaryDark">#3700B3</color>
  <color name="colorAccent">#03DAC5</color>
  <color name="loadingBackground">#80FFFFFF</color>
  <color name="fragmentBackground">#FFFFFF</color>
</resources>
```

Теперь доработаем нашу модель. Нам потребуется передавать данные из одного фрагмента в другой, и для этого мы будем использовать Bundle. Можно передавать в бандл значения из классов City и Weather и восстанавливать их, но это трудоёмкий и многословный процесс. Лучше воспользуемся возможностями языка Kotlin, а именно Kotlin Android Extensions — библиотекой расширений для Android, чтобы сразу передавать parcelable-объект. Для этого пропишем плагин в нашем файле Gradle проекта:

```
apply plugin: 'kotlin-android-extensions'
```

```
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.Androi
    }

    buildFeatures {
        viewBinding true
    }
```

Синхронизируем проект. Теперь посредством простых аннотаций можно парселизовать объекты, не прописывая вспомогательных методов. Дополним классы City и Weather аннотациями и отнаследуемся от соответствующего интерфейса:

```
import android.os.Parcelable
import kotlinx.android.parcel.Parcelize

@Parcelize
data class City(
   val city: String,
```

```
    val lat: Double,
    val lon: Double
) : Parcelable

import android.os.Parcelable
import kotlinx.android.parcel.Parcelize

@Parcelize
data class Weather(
    val city: City = getDefaultCity(),
    val temperature: Int = 0,
    val feelsLike: Int = 0
) : Parcelable
```

Теперь классы можно хранить в бандле. Гораздо лаконичней, чем в Java. Сейчас фрагмент выглядит так:

```
class DetailsFragment : Fragment() {

    private var _binding: FragmentDetailsBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding = FragmentDetailsBinding.inflate(inflater, container, false)
        return binding.getRoot()
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(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()
        }
    }

    companion object {

        const val BUNDLE_EXTRA = "weather"

        fun newInstance(bundle: Bundle): DetailsFragment {
            val fragment = DetailsFragment()
```

```
            fragment.arguments = bundle
            return fragment
        }
    }
}
```

Обратите внимание на companion object. В нём добавилась константа с ключевым словом const, аналог статики в Java, по которой мы будем находить бандл. Немного изменился метод newInstance: теперь туда передаётся бандл с данными о городе. В остальном код стандартен, кроме использования знака вопроса при получении бандла: если бандл == null, то и weather будет == null.

Доработаем MainFragment. Для начала опишем во фрагменте интерфейс, чтобы передавать данные между адаптером списка и фрагментом:

```
interface OnItemViewClickListener {
    fun onItemViewClick(weather: Weather)
}
```

Доработаем MainFragmentAdapter. Будем передавать в его конструктор интерфейс слушателя нажатий и определим метод removeListener, чтобы не возникало утечек памяти. Рекомендуется вызывать этот метод адаптера в методе onDestroy фрагмента MainFragment.

```
class MainFragmentAdapter(private var onItemViewClickListener:
MainFragment.OnItemViewClickListener?)...

fun removeListener() {
    onItemViewClickListener = null
}
```

Осталось вызывать слушатель нажатий по нажатию на элемент:

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

Теперь в MainFragment’е создаём интерфейс и передаём его в адаптер. Обратите внимание, как создаётся интерфейс, — через ключевое слово object, как и рассказывалось на предыдущем занятии.

В самом методе onItemViewClick мы обращаемся к менеджеру фрагментов через активити и создаём бандл. Добавляем в бандл получаемый класс и открываем новый фрагмент:

```
private val adapter = MainFragmentAdapter(object : OnItemViewClickListener {
    override fun onItemViewClick(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()
        }
    }
})
```

Последнее изменение, которое следит за утечками, удаляет слушатель из адаптера:

```
override fun onDestroy() {
    adapter.removeListener()
    super.onDestroy()
}
```

MainFragment полностью:

```
class MainFragment : Fragment() {

    private var _binding: FragmentMainBinding? = null
    private val binding get() = _binding!!

    private lateinit var viewModel: MainViewModel
    private val adapter = MainFragmentAdapter(object : OnItemViewClickListener {
        override fun onItemViewClick(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 var isDataSetRus: Boolean = true

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
```

```
        savedInstanceState: Bundle?
    ): View {
        _binding = FragmentMainBinding.inflate(inflater, container, false)
        return binding.getRoot()
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        binding.mainFragmentRecyclerView.adapter = adapter
        binding.mainFragmentFAB.setOnClickListener { changeWeatherDataSet() }
        viewModel = ViewModelProvider(this).get(MainViewModel::class.java)
        viewModel.getLiveData().observe(viewLifecycleOwner, Observer {
renderData(it) })
        viewModel.getWeatherFromLocalSourceRus()
    }

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

    private fun renderData(appState: AppState) {
        when (appState) {
            is AppState.Success -> {
                binding.mainFragmentLoadingLayout.visibility = View.GONE
                adapter.setWeather(appState.weatherData)
            }
            is AppState.Loading -> {
                binding.mainFragmentLoadingLayout.visibility = View.VISIBLE
            }
            is AppState.Error -> {
                binding.mainFragmentLoadingLayout.visibility = View.GONE
                Snackbar
                    .make(binding.mainFragmentFAB, getString(R.string.error),
Snackbar.LENGTH_INDEFINITE)
                    .setAction(getString(R.string.reload)) {
viewModel.getWeatherFromLocalSourceRus() }
                    .show()
            }
        }
    }

    interface OnItemViewClickListener {
        fun onItemViewClick(weather: Weather)
    }

    companion object {
```

```
        fun newInstance() =
            MainFragment()
    }
}
```

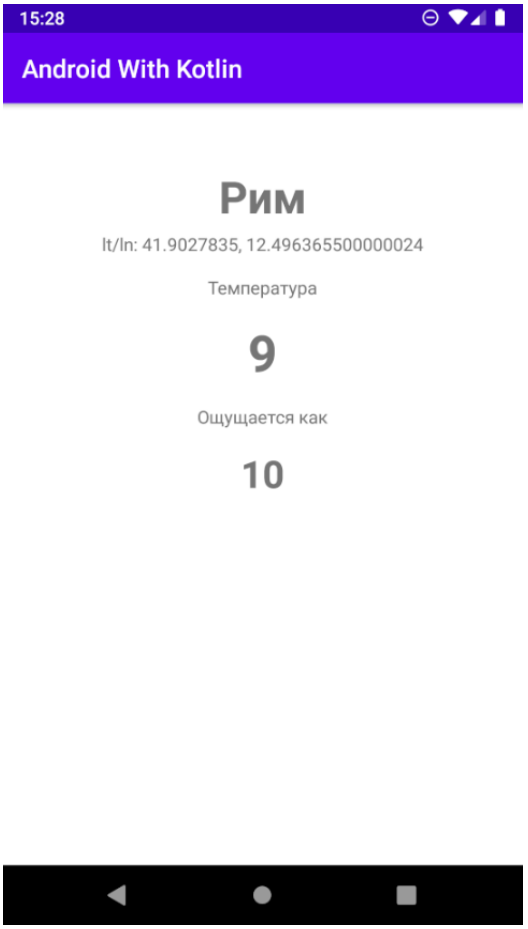
Мы доработали адаптер в соответствии с предыдущими изменениями. Теперь в конструктор передаём листенер. Обратите внимание, что в конце типа стоит знак вопроса. Это говорит о том, что листенер может быть == null.

Добавляем метод для удаления листенера и вызываем его метод через знак вопроса, чтобы обезопасить себя от NPE. Полный код адаптера:

```
class MainFragmentAdapter(private var onItemViewClickListener:
MainFragment.OnItemViewClickListener?) :
    RecyclerView.Adapter<MainFragmentAdapter.MainViewHolder>() {

    private var weatherData: List<Weather> = listOf()

    fun setWeather(data: List<Weather>) {
        weatherData = data
        notifyDataSetChanged()
    }

    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): MainViewHolder {
        return MainViewHolder(
            LayoutInflater.from(parent.context)
                .inflate(R.layout.fragment_main_recycler_item, parent, false) as
View
        )
    }

    override fun onBindViewHolder(holder: MainViewHolder, position: Int) {
        holder.bind(weatherData[position])
    }

    override fun getItemCount(): Int {
        return weatherData.size
    }

    inner class MainViewHolder(view: View) : RecyclerView.ViewHolder(view) {

        fun bind(weather: Weather) {

itemView.findViewById<TextView>(R.id.mainFragmentRecyclerItemTextView).text =
weather.city.city
            itemView.setOnClickListener {
                onItemViewClickListener?.onItemViewClick(weather)
```

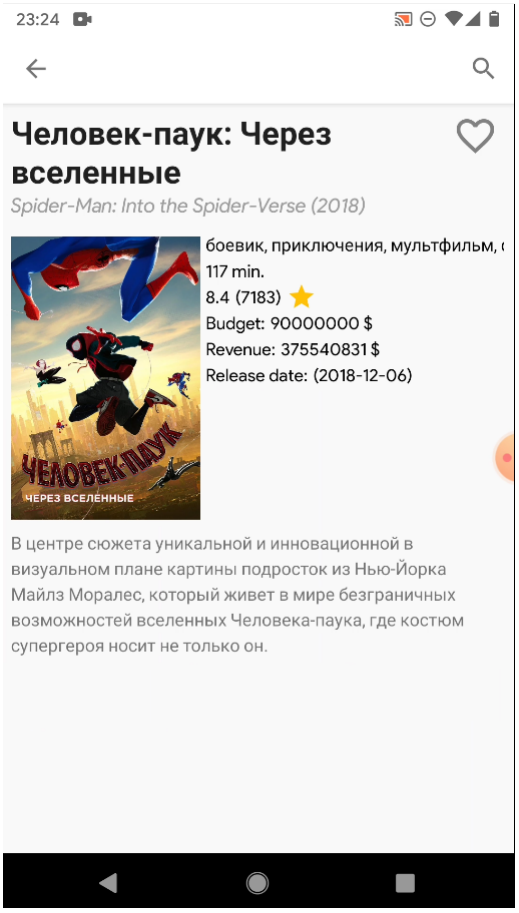
```
                }
            }
        }
    }
```

Запускаем и проверяем:

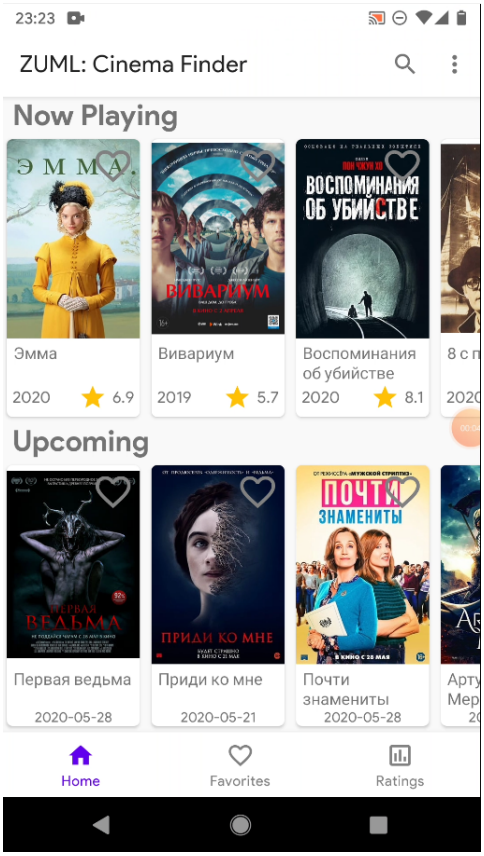


Практическое задание

1. Добавьте экран с описанием конкретного фильма по аналогии с погодным приложением.



2. **Задача для самостоятельного изучения:** добавьте несколько горизонтальных списков для разных категорий фильмов.



Дополнительные материалы

1. Коллекции.
2. Дженерики в Kotlin и Java.

Используемые источники

1. Обобщения (Generics).
2. Дмитрий Жемеров, Светлана Исакова «Kotlin в действии».