ТЕХНОТРЕК

Занятие №3

Современная разработка под Android

Юрий Береза
Кирилл Филимонов

Напоминание

А ты отметился о присутствии на занятии?

Kotlin - Часть 2

1. Классы
2. Интерфейсы и делегирование
3. Ключевое слово Object
4. Объекты-компаньоны
5. Sealed-классы
6. Перегрузка операторов
7. Функции-расширения и Infix-функции
8. Исключения
9. try-with-resources и функция use
10. Коллекции
11. Generics

Классы

- Обыкновенный класс
- Абстрактный класс
- Класс данных (data class)
- Запечатанный класс (sealed class)
- Вложенный класс
- Внутренний класс

Классы

- Открыты для использования (public)
- Закрыты для наследования (final)
- Абстрактные классы по умолчанию открыты для наследования
- Не содержат статических методов
- Внутренние классы не хранят ссылку на внешний класс
- Нет области видимости package private
- Есть область видимости internal

Объявление класса

```
class BasePerson constructor(name : String) {

    val name : String

    init {
        this.name = name
    }
}
```

Объявление класса

- Слово constructor можно убрать, если нет модификатора видимости
- Блок init можно убрать, если свойства объявлять в конструкторе

Проще

```
class BasePerson(name : String) {

    val name : String

    init {
        this.name = name
    }
}
```

Еще проще

```
class BasePerson(val name : String) {

}
```

Конструкторы класса

- Основной конструктор

```
class Person(val name : String)
```

- Вторичный конструктор

```
class Person {
    constructor(name: String) {
        // constructor
    }
}
```

Конструкторы класса

Конструктор по-умолчанию

- не определен ни один конструктор
- все параметры конструктора имеют значение по-умолчанию

```
class User

class NamedUser(val name: String = "Bob")
```

Свойства

```
class Person(name : String) {

    var name : String = name
        get() {
            println("")
            return field
        }
        private set(value) {
            field = value
        }

    val age : Int
        get() {
            return 42
        }
}
```

Модификаторы видимости

private	• Объекты верхнего уровня (классы, функции, интерфейсы, свойства) - доступны только внутри данного файла • Внутри класса - доступны только из этого класса
protected	• Объекты верхнего уровня (классы, функции, интерфейсы, свойства) - не могут иметь такой модификатор • Внутри класса - доступны из этого класса и из наследников класса
internal	• Объекты верхнего уровня (классы, функции, интерфейсы, свойства) - доступны внутри модуля • Внутри класса - доступны внутри модуля, если доступен класс
public	Доступны всем

Два конструктора и более

```
class AgedPerson constructor(name : String, age : Int) {

    private val name : String
    private val age : Int

    constructor(name: String) : this(name, 42)

    constructor(age: Int) : this(age = age, name = "Smith")

    init {
        this.name = name
        this.age = age
    }
}
```

Private

```
//compiler.kt
package com.example.ybereza.helloandroid

private class Lexer {
    private data class Lexem(val token : String)

    private fun parse(text : String) : List<Lexem> {
        return arrayListOf(Lexem("class"), Lexem("space"),
                Lexem("constructor"))
    }

    fun startParsing(text : String) {
        parse(text)
    }
}

private fun compiler(source : String) {
    val lexer = Lexer() // OK - visible inside same file
    val List<Lexer.Lexem> lexems = lexer.parse(source) // Error - private in class

    lexer.startParsing(source) // OK - public method
}
```

Private конструктор

```
// factory
class LocalStorage private constructor() {

    companion object {

        fun get(): LocalStorage {
            // caching here
            return LocalStorage()
        }
    }
}
```

Protected

```
open class BasePerson(name : String) {
    protected val name = name
}

class Person(name: String, val gender : String) : BasePerson(name) {
    fun whoIs() {
        println("I'm $gender and my name is $name")
    }
}
```

Internal

```
internal open class BasePerson(name : String) {
    protected val name = name
}
```

Абстрактный класс

- содержит модификатор класса abstract
- должен быть переопределен
- не может иметь экземпляров
- может содержать abstract методы
- abstract методы не могут иметь реализацию
- может хранить состояние

Абстрактный класс

```
// base presenter
abstract class BasePresenter(val view: View) {

    abstract fun saveState()
    abstract fun restoreState()

    fun onCreate() {
        restoreState()
        // do smth onCreate
    }
}
```

Наследование

- По умолчанию классы закрыты для наследования (**final**)
- Отсутствуют ключевые слова extends и implements
- Ключевое слово **open** разрешает наследование
- Ключевое слово **override** обязательно
- Можно запретить переопределение

Наследование

```
// console logger
open class Logger {

    open fun log(message: String) {
        println(message)
    }

    // public but not open
    fun logToConsole(message: String) {
        println(message)
    }
}

class FileLogger(val file: File): Logger() {

    // overriden, disabled for re-override
    final override fun log(message: String) {
        // log to file
    }
}
```

Интерфейсы

Схожи с интерфейсами в Java 8

- Могут иметь как абстрактные методы, так и реализации
- В отличие от абстрактных классов не хранят состояние
- Могут содержать абстрактные свойства
- Все методы интерфейса **open**
- Если два интерфейса реализуют один и тот же метод, в классе, который их реализует, необходимо будет явно переопределить этот метод.
- Есть возможность указать, какой из методов вызывается

Интерфейсы

```
interface TextWatcher {
    val tag: String

    fun beforeTextChanged(charSequence: CharSequence)
    fun onTextChanged(charSequence: CharSequence)
    fun afterTextChanged(charSequence: CharSequence)
}

class TextWatcherImpl(override val tag: String) : TextWatcher {

    override fun beforeTextChanged(charSequence: CharSequence) {
        //...
    }

    override fun onTextChanged(charSequence: CharSequence) {
        //...
    }

    override fun afterTextChanged(charSequence: CharSequence) {
        //...
    }
}
```

Интерфейсы

```
interface ResourceObservable {
    fun onChange()
    fun onPostChange() = println("resource changed")
}

interface ToggleListener {
    fun onChange()
    fun onPostChange() = println("toggle status changed")
}

class Presenter: ResourceObservable, ToggleListener {

    override fun onChange() {
        println("onChanged called")
    }

    override fun onPostChange() {
        super<ResourceObservable>.onPostChange()
        super<ToggleListener>.onPostChange()
    }
}
```

data классы

- Класс-контейнер для данных (**POJO**)
- Методы equals, hashcode, toString генерируются
- Механизм копирования с частичным изменением (метод **copy**)

Отдавайте предпочтение неизменяемым data классам (**val**)

```
data class Profile(val name: String, val avatarUrl: String, val accountType: Type)
```

Делегирование

- Поддержка подхода “предпочитайте композицию наследованию” на уровне языка
- Реализация шаблона “Декоратор”
- Сокращение количества кода
- Ключевое слово **by**

```
class CountRemovedInterator<T>(val iterator: MutableIterator<T>): MutableIterator<T> by iterator {
    private var removedCount: Int = 0
        private set

    override fun remove() {
        removedCount++
        iterator.remove()
    }
}
```

Вложенные и внутренние классы

Вложенный класс:

- Класс объявленный внутри другого класса
- Не имеет доступа к экземпляру внешнего класса (аналог статического внутреннего класса в Java)

Внутренний класс:

- Класс объявленный внутри другого класса с ключевым словом inner
- Имеет доступ к экземпляру внешнего класса (аналог нестатического внутреннего класса в Java)

Вложенные и внутренние классы

```
class Outer {
    val outer: String = "outer"

  // Вложенный класс
  class Nested {

    fun print() {
        println("Hello $outer from nested") // Error
    }

  }

  inner class Inner {

    fun print() {
        println("Hello $outer from inner") // OK
    }

  }
}
```

sealed классы

- Расширение enum-классов
- Реализуют ограниченную иерархию типов
- В отличие от enum может быть несколько экземпляров типа
- Определение может быть в любом месте
- Работают в конструкции **when** (compile-time проверка)
- Не требуют обработки условия **else**
- Объявляются с помощью ключевого слова **sealed**

sealed классы

```
sealed class UIState {
    class Ok(val message: String, val isLoaded: Boolean): UIState()
    class Error(val errorMessage: String): UIState()
    class Loading(): UIState()
}

class View {

    fun onStateChanged(state: UIState) {
        when(state) {
            is UIState.Ok -> // handle OK State
            is UIState.Error -> // handle Error state
            is UIState.Loading -> // handle Loading state
        }
    }
}
```

Ключевое слово object

- Объявление объекта (Singleton)
- Реализация объекта-компаньона
- Запись объекта-выражения

Object: объявление объекта

- Реализация шаблона Singleton
- Объявление класса + создание экземпляра
- Гарантируется единственность экземпляра
- Может содержать свойства, методы, init
- Может наследовать классы и интерфейсы
- Не может содержать конструкторов

Object: объявление объекта

```
object MyTrackerAnalyticsLogger(private val tracker: MyTracker) {
    override fun logEvent(event: Event) {
        tracker.log(event)
    }
}

fun logInstallEvent() {
    MyTrackerAnalyticsLogger.logEvent(InstallEvent())
}
```

Object: объект-компаньон

- объект внутри класса
- замена статическим методам и членам класса
- имеет доступ к членам класса
- может реализовывать интерфейс
- может иметь функции-расширения
- позволяет реализовать фабричный метод

Object: объект-компаньон

```
class Event private constructor(private val prefix: String, private val type: String) {

    companion object {
        fun create(type: Type): Event = when(type) {
                is Intall -> Event("INS", "Install")
                is Start -> Event("STRT", "Start")
                is Click -> Event("CLK", "Click")
            }
        }
}

fun onAppInstalled() {
    logger.log(Event.create(Event.Install))
}
```

Object: объект-выражение

- создание анонимных объектов
- замена анонимным внутренним классам в Java
- могут реализовывать несколько интерфейсов

```
editText.addTextChangedListener(object : TextWatcher {
    override fun afterTextChanged(s: Editable?) {
        // ..
    }

    override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {
        // ..
    }

    override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
        // ..
    }
})
```

Функции-расширения

- Добавляют новую функциональность классам
- Не меняют его состояния
- Могут быть добавлены в закрытые классы
- В том числе и в библиотечные классы
- Объявляются за пределами этого класса
- Не могут быть переопределены

Функции-расширения

```
fun String.asJSONObject() = try {
        JSONObject(this)
    }
    catch (e : JSONException) {
        JSONObject()
    }

fun test() {
    """{"key" : "value"}""".asJSONObject().getString("key")?.let {
        if (it == "value") {
            println("found value!")
        }
    }
}
```

Функции-расширения infix-нотация

```kotlin
infix fun String.join(next : String) : String = this.plus(next)

fun example() {
    val sentence = "Hello" join ", " join "World!"
    println(sentence) // Hello, World!
}
```

Функции let и apply

```
class MyClass {
    val myString = "My String"
    var myInt = 3

    fun call() {
        val substring = myString.let {
            it.substring(myInt)
        }
        println(substring) // String

        val sbs : String = myString.apply {
            substring(myInt)
        }
        println(sbs) // My String
    }
}
```

Перегрузка операторов

- Ограниченный набор операторов
- Для перегрузки используется слово **`operator`**
- Может быть как член класса
- Так и функцией-расширением
- При переопределении используется ключевое слово, а не сам оператор

Перегрузка операторов - имена функций

a++	a.inc()
a--	a.dec()
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a..b	a.rangeTo(b)
a[i]	a.get(i)
a[i] = b	a.set(i, b)
a += b	a.plusAssign(b)
a < b, a > b, a >= b, a <= b	a.compareTo(b)
a == b, a != b	a?.equals(b)

Перегрузка арифметических операторов

```
data class Point(val x : Int, val y : Int) {
    operator fun plus(other : Point) = Point(x + other.x, y + other.y)
}

val p1 = Point(10, 10)
val p2 = Point(10, 15)
val p3 = p1 + p2 // Point(20, 25)
```

Перегрузка операторов equals

```
class Point(val x : Int, val y : Int) {
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other !is Point) return false
        return Objects.equals(x, other.x) && Objects.equals(y, other.y)
    }
}

val p1 = Point(10, 10)
val p2 = Point(10, 10)
val p3 = p1 == p2 // true
val p4 = p1 != p2 // false
```

Перегрузка операторов compareTo

```
data class Person(val name : String, val age : Int) : Comparable<Person> {
    override fun compareTo(other: Person) : Int {
        return compareValuesBy(this, other, Person::age)
    }
}

fun main() {
    val p1 = Person("Alice", 23)
    val p2 = Person("Bob", 29)

    println(p1 <= p2) // true
}
```

Исключения

- Механизм порождения и перехвата аналогичен Java
- Все исключения обрабатываются как Unchecked
- Отсутствует механизм try-with-resources
- Однако есть замена
- try-catch - это выражение

Исключения

```
/**
 * This function parses line as Int
 * @throws IllegalStateException if line is not a number
 */
fun ReadNumber(reader : BufferedReader) : Int {
  try {
    return Integer.parseInt(reader.readLine())
  }
  catch(e: NumberFormatException) {
    throw IllegalStateException("Can not parse data!", e)
  }
}
```

try-выражение

```
/**
 * This function parses line as Int
 * @return number or default value
 */
fun readNumber(reader : BufferedReader, value : Int = 0) : Int {
    val number = try {
        Integer.parseInt(reader.readLine())
    }
    catch(e: NumberFormatException) {
        value
    }
    return number
}
```

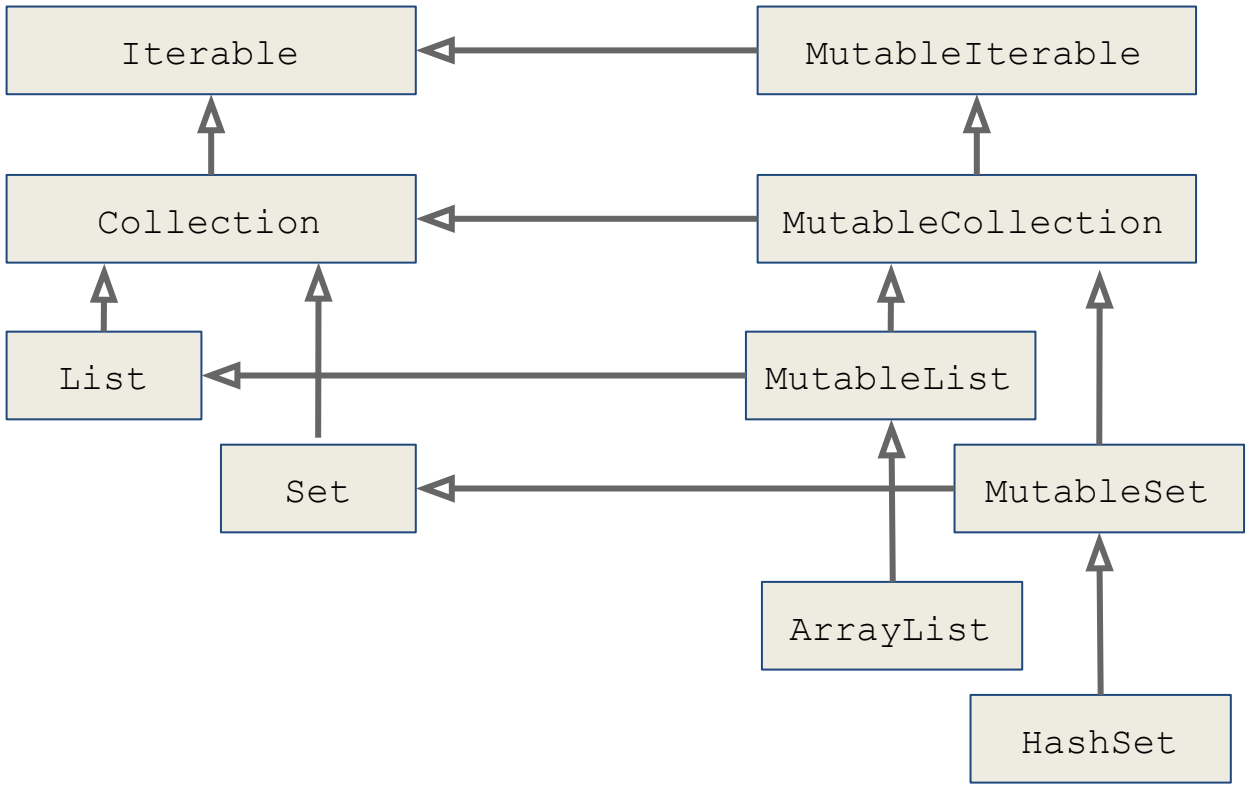
Функция-расширение use

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =
            new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

```
fun readFirstLineFromFile(path : String) =
        BufferedReader(FileReader(path)).use {
            readLine()
        }
```

Коллекции

- Используется библиотека коллекций из Java
- Классы дополнены функциями-расширениями
- Интерфейсы для mutable и immutable коллекций



Коллекции

Создание коллекций

Тип	Immutable	Mutable
`List`	`listOf`	`mutableListOf, arrayListOf`
`Set`	`setOf`	`mutableSetOf, hashSetOf, linkedSetOf, sortedSetOf`
`Map`	`mapOf`	`mutableMapOf, hashMapOf, linkedMapOf, sortedMapOf`

Коллекции

```kotlin
val numbers = listOf(1, 2, 3)
println(numbers[1]) //2
numbers[0] = 3 // Error

val mutableNumbers = mutableListOf(1, 2, 3)
mutableNumbers[0] = 2
println(mutableNumbers[0]) // 2

val map = mapOf(1 to "Hello", 2 to "Kotlin")
for ((key, value) in map) {
    print("$key: $value")
}
```

Массивы

- Класс с параметром типа
- По-умолчанию массив объектов ссылочного типа
- Есть классы для создания массивов примитивных типов

Массивы

Создание массива:

- `arrayOf`
- `arrayOfNulls`
- `Array<T>(size: Int, f: (Int) -> T)`

Создание массива примитивных типов

- `IntArray(size: Int)`, `ByteArray(size: Int)` etc…
- `intArrayOf`, `byteArrayOf`, etc…
- `IntArray(size: Int, f: (Int) -> Int)`, etc...

Массивы

```
val integerArray = Array(5, {i ->  i + 1})
for (i in integerArray) {
    print("$i ") // 1, 2, 3, 4, 5
}

val intArray = IntArray(5)
for (i in intArray.size downTo 0) {
    intArray[i] = i
}
```

Операции над коллекциями

- Операции агрегирования

  `any, all, count, fold, reduce, sumBy, etc`

- Операции фильтрации

  `drop, filter, slice, take, etc`

- Операции отображения

  `map, flatMap, groupBy, mapIndexed, mapNotNull`

- Операции с элементами

  `contains, elementAt, last, first, single, etc`

- Упорядочивание

  `sort, reverse, etc`

Filter

- Фильтрует коллекцию по предикату
- Возвращает новую коллекцию в результате

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

```
val numbers = listOf(1, 2, 3, 4, 5, 6, 7)
val evenNumbers = numbers.filter { it % 2 == 0 }
```

Map и FlatMap

Map

- применяет функцию к каждому элементу
- возвращает новую коллекцию с результатом

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

```
val numbers = listOf(1, 2, 3, 4, 5, 6, 7)
val evenNumbers = numbers.filter { it % 2 == 0 }.map { it * it }
```

Map и FlatMap

FlatMap

- преобразует каждый элемент в коллекцию
- возвращает в результате объединенную коллекцию

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

```
class Book(val title: String, val authors: List<String>)

fun getAuthors(books: List<Book>): Collection<String> = books.flatMap { it.authors }.toSet()
```

Generics

- Параметризованные функции и свойства
- Параметризованные классы
- Ограничения типов
- Стирание типов (type erasure)
- Овеществляемые (reified) типы
- Вариантность

Generics: функции и классы

```
class Service<Req, Rep>(private val executor: RequestExecutor<Rep, Req>) {

    fun apply(request: Req): Future<Rep> {
        return executor.execute(request)
    }

    fun <T> convert(converter: (Req) -> T): List<T> {
        val list = mutableListOf<T>()
        for (request in executor.requests) {
            list.add(converter(request))
        }
        return list
    }
}
```

Generics

- Тип B - подтип типа A, если значение типа B можно использовать везде, где ожидается значение типа A
- Если B - подтип типа A, то A - супертип для B

Особенность типов в Kotlin

- Int - **является** подтипом Int?
- Int? - **не является** подтипом Int

Generics

- Обобщенный класс C называют **инвариантным** по типовому параметру, если для любых разных типов A и B, C<A> не является подтипом C<B>
- Обобщенный класс C называют **ковариантным** по типовому параметру, если для типов A и B, где A - подтип B, C<A> является подтипом C<B>
- Обобщенный класс C называют **контравариантным** по типовому параметру, если для типов A и B, где B - супертип A, C<B> является подтипом C<A>

Generics: ковариантность

- отношение тип-подтип сохраняется
- типовой параметр используется только на исходящей позиции (ключевое слово **out**)

```
interface List<out T>: Collection<T>  {
    operator fun get(index: Int): T

    //...
}
```

Generics: контравариантность

- отношение тип-подтип изменяется на противоположное
- ключевой параметр используется на входящей позиции (ключевое слово **in**)

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

ТЕХНОТРЕК

Спасибо за внимание!

Юрий Береза
Кирилл Филимонов