Бесплатная электронная книга

учусь Kotlin

Free unaffiliated eBook created from **Stack Overflow contributors.**

#kotlin

. . . 1

1: Kotlin . . . 2

. . . 2

. . . 2

. . . 2

Examples . . . 3

, . . . 3

, , . . . 4

Hello World, Companion . . . 4

varargs . . . 5

. . . 5

. . . 5

2: Enum . . . 7

. . . 7

Examples . . . 7

. . . 7

. . . 7

. . . 8

. . . 8

3: JUnit . . . 9

Examples . . . 9

. . . 9

4: Kotlin Android Extensions . . . 10

. . . 10

Examples . . . 10

. . . 10

. . . 10

. . . 11

, . . . 12

5: Kotlin Java . . . 13

. . . 13

Examples 13
13
13
14
IF, TRY , 14
6: RecyclerView 15
15
Examples 15
15
7: Regex 17
Examples 17
17
: 17
: 17
: 17
18
RegEx 18
18
19
find (input: CharSequence, startIndex: Int): MatchResult? 19
findAll (: CharSequence, startIndex: Int): 20
matchEntire (: CharSequence): MatchResult? 20
match (input: CharSequence): Boolean 20
containsMatchIn (: CharSequence): Boolean 21
split (: CharSequence, limit: Int): 21
replace (input: CharSequence, replacement: String): String 21
8: 23
Examples 23
23
- 23
9: 25

25
25
Examples 26
26
26
26
10: 27
27
Examples 27
: vararg 27
: vararg 27
11: 29
29
Examples 29
29
12: 30
30
Examples 30
30
30
30
30
31
13: 33
33
33
33
33
- Nullable 33
Examples 34
34
34

14: **36**
Examples 36
DTO (POJO / POCOs) 36
..... 36
, ... 36
serialVersionUid 37
..... 38
let 38
..... 39
15: **40**
..... 40
Examples 40
..... 40
downTo () 40
step () 40
..... 40
16: **42**
..... 42
Examples 42
..... 42
..... 42
..... 42
..... 43
..... 43
..... 44
..... 44
17: **46**
Examples 46
try-catch-finally 46
18: **47**
..... 47
..... 47

Examples 47
47
47
47
19: 48
Examples 48
48
48
49
49
49
50
50
20: 51
51
51
Examples 51
51
Pitfall: 51
, , 52
Java 7+ Path class 52
53
Java 8 ISO 53
- ( ) 54
54
54
55
55
21: 56
56
56
Examples 56

56

22: **57**

.... 57

.... 57

.... 57

Examples 57

: 'open' 57

.... 58

: 58

: 58

: 58

.... 58

: 58

: 58

Person 59

.... 59

(,): 59

: 59

23: Kotlin **61**

Examples 61

.... 61

JVM 61

Android 61

JS 61

Android Studio 62

.... 62

.... 62

Java 63

Gradle Groovy Kotlin 63

24: **65**

Examples 65

Nullable Non-Nullable 65

.... 65

: 65

- 66

Iterable 66

Null Coalescing / Elvis Operator 66

.... 67

(? :) 67

25: Singleton 69

.... 69

Examples 69

/ java 69

.... 69

26: 71

.... 71

.... 71

Examples 71

.... 71

27: 73

.... 73

.... 73

Examples 73

.... 73

.... 73

Java 73

.... 74

.... 75

28: 77

.... 77

Examples 77

x 77

.... 77

.... 78

.... 78
.... 78
.... 79
.... 79
29: **80**
Examples 80
toString () NULL 80
30: **81**
.... 81
Examples 81
kotlin.logging 81
31: **82**
.... 82
Examples 82
, 1 , ... 82
32: DSL **83**
.... 83
Examples 83
Infix DSL 83
DSL 83
.... 83
lambdas 84
33: **85**
Examples 85
.... 85
.... 85
.... 86
.... 86
34: **88**
.... 88
.... 88
.... 88

Examples ... 88
... 88
... 88
35: - ... 89
... 89
... 89
- ... 89
Examples ... 89
- ... 89
36: ... 91
... 91
Examples ... 91
if-statement ... 91
if ... 91
when if-else-if ... 92
- ... 92
- ... 93
- ... 93
37: ... 95
... 95
... 95
Examples ... 95
, ... 95
- ... 96
... 97
... 98
... 99
... 99
... 99
38: Java 8 ... 101
... 101
... 101

.... 101
?!? 101
.... 102
: 103
Examples 103
.... 103
, ... 103
.... 103
.... 103
.... 104
.... 104
- 104
.... 104
.... 105
.... 105
2 - , ... 105
3 - 105
4 - , , ... 105
5 - , , ... 106
6 - , , ... 106
7 - , Int, ... 106
.... 107
- , , ... 107
1 - , ... 107
5 - , , ... 108
6 - , ... 108
7a - , ... 109
7b - SummarizingInt 110
.... **112**

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: kotlin

It is an unofficial and free Kotlin ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Kotlin.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с Kotlin

замечания

Kotlin - это статически типизированный объектно-ориентированный язык программирования, разработанный JetBrains, ориентированный прежде всего на JVM. Kotlin разработан с целями быстрого компиляции, обратной совместимости, безопасного типа и 100% совместимости с Java. Kotlin также разработан с целью предоставления многих функций, которые хотят разработчики Java. Стандартный компилятор Kotlin позволяет скомпилировать его как в байт-код Java для JVM, так и в JavaScript.

Компиляция Котлина

У Kotlin есть стандартный плагин IDE для Eclipse и IntelliJ. Kotlin также может быть скомпилирован с использованием Maven , используя Ant , и используя Gradle или через командную строку .

Стоит отметить, что в `$ kotlinc Main.kt` выведет файл класса java, в данном случае `MainKt.class` (обратите внимание, что Kt добавлено к имени класса). Однако, если кто-то должен был запускать файл класса с помощью `$ java MainKt` java выдаст следующее исключение:

```
Exception in thread "main" java.lang.NoClassDefFoundError: kotlin/jvm/internal/Intrinsics
    at MainKt.main(Main.kt)
Caused by: java.lang.ClassNotFoundException: kotlin.jvm.internal.Intrinsics
    at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:335)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    ... 1 more
```

Чтобы запустить результирующий файл класса с помощью Java, необходимо включить файл jar-файла времени выполнения Kotlin в текущий путь класса.

```
java -cp .:/path/to/kotlin/runtime/jar/kotlin-runtime.jar MainKt
```

Версии

Версия	Дата выхода
1.0.0	2016-02-15
1.0.1	2016-03-16

Версия	Дата выхода
1.0.2	2016-05-13
1.0.3	2016-06-30
1.0.4	2016-09-22
1.0.5	2016-11-08
1.0.6	2016-12-27
1.1.0	2017-03-01
1.1.1	2017-03-14
1.1.2	2017-04-25
1.1.3	2017-06-23

Examples

Привет, мир

Все программы Kotlin начинаются с `main` функции. Вот пример простой программы Kotlin «Hello World»:

```
package my.program

fun main(args: Array<String>) {
    println("Hello, world!")
}
```

Поместите вышеуказанный код в файл с именем `Main.kt` (это имя файла полностью произвольно)

При ориентации на JVM функция будет скомпилирована как статический метод в классе с именем, полученным из имени файла. В приведенном выше примере основным классом для запуска будет `my.program.MainKt` .

Чтобы изменить имя класса, содержащего функции верхнего уровня для определенного файла, поместите следующую аннотацию в верхней части файла над оператором пакета:

```
@file:JvmName("MyApp")
```

В этом примере основной класс для запуска теперь будет `my.program.MyApp` .

Смотрите также:

- Функции уровня пакета, включая аннотацию `@JvmName` .
- Цели использования целевых объектов аннотации

Привет, мир, используя декларацию объекта

Вы также можете использовать декларацию объекта, которая содержит основную функцию для программы Kotlin.

```
package my.program

object App {
    @JvmStatic fun main(args: Array<String>) {
        println("Hello World")
    }
}
```

Имя класса, которое вы запустите, - это имя вашего объекта, в этом случае `my.program.App` .

Преимущество этого метода над функцией верхнего уровня заключается в том, что имя класса для запуска более самоочевидно, а любые другие функции, которые вы добавляете, попадают в класс `App` . У вас также есть экземпляр Singleton `App` для хранения состояния и выполнения другой работы.

Смотрите также:

- Статические методы, включая аннотацию `@JvmStatic`

Hello World, используя объект Companion

Подобно использованию Object Object, вы можете определить `main` функцию программы Kotlin, используя объект Companion класса.

```
package my.program

class App {
    companion object {
        @JvmStatic fun main(args: Array<String>) {
            println("Hello World")
        }
    }
}
```

Имя класса, которое вы запустите, - это имя вашего класса, в этом случае `my.program.App` .

Преимущество этого метода над функцией верхнего уровня заключается в том, что имя класса для запуска более самоочевидно, а любые другие функции, которые вы добавляете, попадают в класс `App` . Это похоже на пример `Object Declaration` , кроме того, что вы контролируете создание экземпляров любых классов для дальнейшей работы.

Небольшое отклонение, которое создает класс для действительного «привет»:

```
class App {
    companion object {
        @JvmStatic fun main(args: Array<String>) {
            App().run()
        }
    }

    fun run() {
        println("Hello World")
    }
}
```

Смотрите также:

- Статические методы, включая аннотацию @JvmStatic

Основные методы с использованием varargs

Все эти основные стили методов также могут использоваться с varargs :

```
package my.program

fun main(vararg args: String) {
    println("Hello, world!")
}
```

Компилировать и запускать код Котлина в командной строке

Поскольку java предоставляет две разные команды для компиляции и запуска кода Java. То же, что и Котлин, также предоставляет вам разные команды.

`javac` для компиляции java-файлов. `java` для запуска java-файлов.

То же, что `kotlinc` для компиляции kotlin файлов `kotlin` для запуска kotlin файлов.

Чтение ввода из командной строки

Аргументы, переданные с консоли, могут быть получены в программе Kotlin и могут использоваться как входные данные. Вы можете передать N (1 2 3 и так далее) количество аргументов из командной строки.

Простой пример аргумента командной строки в Котлине.

```
fun main(args: Array<String>) {

    println("Enter Two number")
    var (a, b) = readLine()!!.split(' ') // !! this operator use for
NPE(NullPointerException).

```

```
    println("Max number is : ${maxNum(a.toInt(), b.toInt())}")
}



fun maxNum(a: Int, b: Int): Int {

    var max = if (a > b) {
        println("The value of a is $a");
        a
    } else {
        println("The value of b is $b")
        b
    }

    return max;

}
```

Здесь введите два числа из командной строки, чтобы найти максимальное число. Выход :

```
Enter Two number
71 89 // Enter two number from command line

The value of b is 89
Max number is: 89
```

За !! Оператор Пожалуйста, проверьте Null Safety .

Примечание: выше пример компиляции и запуска на Intellij.

Прочитайте Начало работы с Kotlin онлайн: https://riptutorial.com/ru/kotlin/topic/490/начало-работы-с-kotlin

глава 2: Enum

замечания

Как и в Java, классы enum в Kotlin имеют синтетические методы, позволяющие перечислять определенные константы enum и получать константу enum по ее имени. Подписи этих методов следующие (при условии, что имя класса enum - `EnumClass`):

```
EnumClass.valueOf(value: String): EnumClass
EnumClass.values(): Array<EnumClass>
```

Метод `valueOf()` `IllegalArgumentException` если указанное имя не соответствует ни одной из констант перечисления, определенных в классе.

Каждая константа перечисления имеет свойства для получения своего имени и позиции в объявлении класса enum:

```
val name: String
val ordinal: Int
```

Константы перечисления также реализуют интерфейс Comparable, при этом естественным порядком является порядок, в котором они определены в классе enum.

Examples

инициализация

Классы Enum, как и любые другие классы, могут иметь конструктор и быть инициализированы

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```

Функции и свойства в перечислениях

Классы enum могут также объявлять члены (т. Е. Свойства и функции). Точка с запятой (`;`) должна быть помещена между последним объектом перечисления и объявлением первого члена.

Если элемент является `abstract` , объекты перечисления должны его реализовать.

```
enum class Color {
    RED {
        override val rgb: Int = 0xFF0000
    },
    GREEN {
        override val rgb: Int = 0x00FF00
    },
    BLUE {
        override val rgb: Int = 0x0000FF
    }

    ;

    abstract val rgb: Int

    fun colorString() = "#%06X".format(0xFFFFFF and rgb)
}
```

Простой перечисление

```
enum class Color {
  RED, GREEN, BLUE
}
```

Каждая константа перечисления является объектом. Константы континуума разделяются запятыми.

переменчивость

Перечисления могут быть изменчивыми, это еще один способ получить одноэлементное поведение:

```
enum class Planet(var population: Int = 0) {
    EARTH(7 * 100000000),
    MARS();

    override fun toString() = "$name[population=$population]"
}

 println(Planet.MARS) // MARS[population=0]
 Planet.MARS.population = 3
 println(Planet.MARS) // MARS[population=3]
```

Прочитайте Enum онлайн: https://riptutorial.com/ru/kotlin/topic/2286/enum

глава 3: JUnit

Examples

правила

Чтобы добавить правило JUnit к тестовому устройству:

```
@Rule @JvmField val myRule = TemporaryFolder()
```

Аннотации `@JvmField` необходимы для `@JvmField` поля поддержки с той же видимостью (public), что и свойство `myRule` (см. Ответ). Правила JUnit требуют, чтобы поле аннотированного правила было общедоступным.

Прочитайте JUnit онлайн: https://riptutorial.com/ru/kotlin/topic/6973/junit

глава 4: Kotlin Android Extensions

Вступление

У Kotlin есть встроенная вставка для Android, позволяющая пропускать ручную привязку или необходимость в таких фреймворках, как ButterKnife. Некоторые из преимуществ - более хороший синтаксис, более статическая типизация и, следовательно, менее подверженная ошибкам.

Examples

конфигурация

Начните с правильно настроенного проекта градиента .

В своем **проекте-локальном** (а не на верхнем уровне) `build.gradle` добавьте объявление плагина расширения под вашим плагином Kotlin на уровне отступа верхнего уровня.

```
buildscript {
    ...
}

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

Использование представлений

Предполагая, что у нас есть активность с примером макета под названием `activity_main.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/my_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="My button"/>
</LinearLayout>
```

Мы можем использовать расширения Kotlin для вызова кнопки без дополнительной привязки, например:

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

class MainActivity: Activity() {
    override fun onCreate(savedInstanceBundle: Bundle?) {
        super.onCreate(savedInstanceBundle)
        setContentView(R.layout.activity_main)
        // my_button is already casted to a proper type of "Button"
        // instead of being a "View"
        my_button.setText("Kotlin rocks!")
    }
}
```

Вы также можете импортировать все идентификаторы, появляющиеся в макете, с помощью `*` нотации

```
// my_button can be used the same way as before
import kotlinx.android.synthetic.main.activity_main.*
```

Синтетические представления нельзя использовать за пределами Работы / Фрагменты / Представления с раздутым макетом:

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

class NotAView {
    init {
        // This sample won't compile!
        my_button.setText("Kotlin rocks!")
    }
}
```

Ароматизаторы продуктов

Расширения Android также работают с несколькими продуктами Android Product Flavors. Например, если у нас есть `build.gradle` в `build.gradle` вот так:

```
android {
    productFlavors {
        paid {
            ...
        }
        free {
            ...
        }
    }
}
```

И, например, только бесплатный вкус имеет кнопку покупки:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

```

```
    <Button
        android:id="@+id/buy_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Buy full version"/>
</LinearLayout>
```

Мы можем напрямую связываться с ароматом:

```
import kotlinx.android.synthetic.free.main_activity.buy_button
```

Болезненный слушатель для получения уведомления, когда представление полностью нарисовано сейчас, настолько просто и удивительно с расширением Котлина

```
mView.afterMeasured {
  // inside this block the view is completely drawn
  // you can get view's height/width, it.height / it.width
}
```

Под капотом

```
inline fun View.afterMeasured(crossinline f: View.() -> Unit) {
viewTreeObserver.addOnGlobalLayoutListener(object : ViewTreeObserver.OnGlobalLayoutListener {
    override fun onGlobalLayout() {
        if (measuredHeight > 0 && measuredWidth > 0) {
            viewTreeObserver.removeOnGlobalLayoutListener(this)
            f()
        }
    }
})
}
```

Прочитайте Kotlin Android Extensions онлайн: https://riptutorial.com/ru/kotlin/topic/9474/kotlin-android-extensions

глава 5: Kotlin для разработчиков Java

Вступление

У большинства людей, приезжающих в Котлин, есть язык программирования на Java.

В этом разделе собраны примеры, сравнивающие Java с Kotlin, выделяя наиболее важные отличия и те драгоценные камни, которые предлагает Kotlin по Java.

Examples

Объявление переменных

В Kotlin объявления переменных выглядят немного иначе, чем Java:

```
val i : Int = 42
```

- Они начинаются с `val` или `var` , делая `final` декларацию (« **val** ue») или **var** iable.
- Тип указывается после имени, разделенного `:`
- Благодаря выходу *типа* Kotlin явное объявление типа может быть опущено, если есть назначение с типом, который компилятор может однозначно обнаружить

Джава	Котлин
`int i = 42;`	`var i = 42` (*или* `var i : Int = 42`)
`final int i = 42;`	`val i = 42`

Быстрые факты

- Котлин не нужен `;` заканчивать заявления
- Котлин является **нулевым**
- Kotlin на **100% совместим с Java**
- У Kotlin **нет примитивов** (но, если это возможно, оптимизирует их копии объектов для JVM)
- Классы Котлина имеют **свойства, а не поля**
- Kotlin предлагает **классы данных** с автогенерированными методами `equals` / `hashCode` и аксессуарами для полей
- Kotlin только имеет время выполнения Исключения, **не проверено Исключения**
- У Kotlin **нет `new` ключевого слова** . Создание объектов выполняется просто путем вызова конструктора, как и любого другого метода.

- Kotlin поддерживает (ограничивает) **перегрузку оператора** . Например, доступ к значению карты можно записать так: `val a = someMap["key"]`
- Kotlin не только может быть скомпилирован в байтовый код для JVM, но также и в **Java Script** , позволяя вам писать код и код интерфейса в Kotlin
- Kotlin **полностью совместим с Java 6** , что особенно интересно в отношении поддержки (не очень) старых Android-устройств
- Kotlin - **официально поддерживаемый** язык **для разработки Android**
- Коллекции Котлина имеют встроенное разделение между **изменчивыми и неизменными коллекциями** .
- Котлин поддерживает **Coroutines** (экспериментальный)

Равноправие и идентичность

Котлин использует `==` для равенства (т. `equals` Вызовы `equals` внутренне) и `===` для ссылочной идентичности.

Джава	Котлин
`a.equals(b);`	`a == b`
`a == b;`	`a === b`
`a != b;`	`a !== b`

См. Https://kotlinlang.org/docs/reference/equality.html.

IF, TRY и другие выражения, а не выражения

В Kotlin, `if` , `try` и другие выражения (так они возвращают значение), а не (void).

Так, например, у Kotlin нет трехъядерного *оператора Elvis от* Java, но вы можете написать что-то вроде этого:

```
val i = if (someBoolean) 33 else 42
```

Еще более незнакомым, но в равной степени выразительным, является *выражение* `try` :

```
val i = try {
    Integer.parseInt(someString)
}
catch (ex : Exception)
{
    42
}
```

Прочитайте Kotlin для разработчиков Java онлайн:
https://riptutorial.com/ru/kotlin/topic/10099/kotlin-для-разработчиков-java

глава 6: RecyclerView в Котлине

Вступление

Я просто хочу поделиться своими небольшими знаниями и кодом RecyclerView с помощью Kotlin.

Examples

Основной класс и адаптер

Я предполагаю, что вы знаете о каком-то синтаксисе **Kotlin** и о том, как использовать, просто добавьте **RecyclerView** в файл **activity_main.xml** и установите класс адаптера.

```
class MainActivity : AppCompatActivity(){

        lateinit var mRecyclerView : RecyclerView
        val mAdapter : RecyclerAdapter = RecyclerAdapter()

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_main)
            val toolbar = findViewById(R.id.toolbar) as Toolbar
            setSupportActionBar(toolbar)

            mRecyclerView = findViewById(R.id.recycler_view) as RecyclerView

            mRecyclerView.setHasFixedSize(true)
            mRecyclerView.layoutManager = LinearLayoutManager(this)
            mAdapter.RecyclerAdapter(getList(), this)
            mRecyclerView.adapter = mAdapter
        }

        private fun getList(): ArrayList<String> {
            var list : ArrayList<String> = ArrayList()
            for  (i in 1..10) { // equivalent of 1 <= i && i <= 10
                println(i)
                list.add("$i")
            }
            return list
        }
    }
```

это ваш класс **адаптера** просмотра recycler и создайте файл **main_item.xml,** что вы хотите

```
class RecyclerAdapter : RecyclerView.Adapter<RecyclerAdapter.ViewHolder>() {

    var mItems: ArrayList<String>  = ArrayList()
    lateinit var mClick : OnClick

    fun RecyclerAdapter(item : ArrayList<String>, mClick : OnClick){
```

```
        this.mItems = item
        this.mClick = mClick;
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val item = mItems[position]
        holder.bind(item, mClick, position)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val layoutInflater = LayoutInflater.from(parent.context)
        return ViewHolder(layoutInflater.inflate(R.layout.main_item, parent, false))
    }

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

    class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        val card = view.findViewById(R.id.card) as TextView
        fun bind(str: String, mClick: OnClick, position: Int){
            card.text = str
            card.setOnClickListener { view ->
                mClick.onClickListner(position)
            }
        }
    }
}
```

Прочитайте RecyclerView в Котлине онлайн:
https://riptutorial.com/ru/kotlin/topic/10143/recyclerview-в-котлине

глава 7: Regex

Examples

Идиомы для соответствия регулярных выражений при выражении

Использование неизменяемых местных жителей:

Использует меньше горизонтального пространства, но больше вертикального пространства, чем шаблон «анонимных временных». Предпочтительно по шаблону «анонимные временные», если выражение `when` находится в цикле - в этом случае определения регулярных выражений должны быть помещены вне цикла.

```
import kotlin.text.regex

var string = /* some string */

val regex1 = Regex( /* pattern */ )
val regex2 = Regex( /* pattern */ )
/* etc */

when {
    regex1.matches(string) -> /* do stuff */
    regex2.matches(string) -> /* do stuff */
    /* etc */
}
```

Использование анонимных времен:

Использует меньше вертикального пространства, но больше горизонтального пространства, чем шаблон «неизменяемых локалей». Не следует использовать, если тогда, `when` выражение находится в цикле.

```
import kotlin.text.regex

var string = /* some string */

when {
    Regex( /* pattern */ ).matches(string) -> /* do stuff */
    Regex( /* pattern */ ).matches(string) -> /* do stuff */
    /* etc */
}
```

Использование шаблона посетителя:

Имеет преимущество тесного подражания «аргументации» `when` синтаксисе. Это полезно, потому что оно более четко указывает аргумент выражения `when` , а также исключает некоторые ошибки программиста, которые могут возникнуть из-за повторения аргумента `when` в каждом из `whenEntry` . Либо «неизменяемые местные жители», либо шаблон «анонимные временные» могут использоваться с этой реализацией шаблона посетителя.

```
import kotlin.text.regex

var string = /* some string */

when (RegexWhenArgument(string)) {
    Regex( /* pattern */ ) -> /* do stuff */
    Regex( /* pattern */ ) -> /* do stuff */
    /* etc */
}
```

А минимальное определение класса - оболочки для `when` это выражения аргумента:

```
class RegexWhenArgument (val whenArgument: CharSequence) {
    operator fun equals(whenEntry: Regex) = whenEntry.matches(whenArgument)
    override operator fun equals(whenEntry: Any?) = (whenArgument == whenEntry)
}
```

Введение в регулярные выражения в Котлине

В этом сообщении показано, как использовать большинство функций в классе `Regex` , работать с нулевым безопасным образом, связанным с функциями `Regex` , и как сырые строки упрощают запись и чтение шаблонов регулярных выражений.

Класс RegEx

Чтобы работать с регулярными выражениями в Kotlin, вам нужно использовать класс `Regex(pattern: String)` и вызывать такие функции, как `find(..)` или `replace(..)` на этот объект регулярного выражения.

Пример использования класса `Regex` который возвращает true, если строка `input` содержит c или d:

```
val regex = Regex(pattern = "c|d")
val matched = regex.containsMatchIn(input = "abc")    // matched: true
```

Существенное значение для всех функций `Regex` заключается в том, что результат основан на сопоставлении `pattern` регулярного выражения и `input` строки. Некоторые из функций требуют полного соответствия, в то время как для остальных требуется только частичное совпадение. Функция `containsMatchIn(..)` используемая в этом примере, требует частичного соответствия и объясняется позже в этом сообщении.

Нулевая безопасность с регулярными выражениями

Оба `find(..)` и `matchEntire(..)` вернут `MatchResult?` объект. `?` символ после `MatchResult` необходим, чтобы Kotlin `MatchResult` с нулем .

Пример, демонстрирующий, как Kotlin безошибочно обрабатывает значение null из функции `Regex` , когда функция `find(..)` возвращает значение null:

```
val matchResult =
    Regex("c|d").find("efg")           // matchResult: null
val a = matchResult?.value             // a: null
val b = matchResult?.value.orEmpty()   // b: ""
a?.toUpperCase()                       // Still needs question mark. => null
b.toUpperCase()                        // Accesses the function directly. => ""
```

С помощью функции `orEmpty()` `b` не может быть нулевой и `?` символ не нужен, когда вы вызываете функции на `b` .

Если вы не заботитесь об этом безопасном обращении с нулевыми значениями, Kotlin позволяет работать с нулевыми значениями, например, на Java с помощью `!!` персонажи:

```
a!!.toUpperCase()                      // => KotlinNullPointerException
```

Необработанные строки в шаблонах регулярных выражений

Kotlin обеспечивает улучшение по сравнению с Java с помощью необработанной строки, которая позволяет писать чистые шаблоны регулярных выражений без двойных обратных косых черт, которые необходимы с помощью строки Java. Необработанная строка представлена тройной цитатой:

```
"""\d{3}-\d{3}-\d{4}""" // raw Kotlin string
"\\d{3}-\\d{3}-\\d{4}"  // standard Java string
```

find (input: CharSequence, startIndex: Int): MatchResult?

Строка `input` будет сопоставлена с `pattern` в объекте `Regex` . Он возвращает `Matchresult?` объект с первым согласованным текстом после `startIndex` или `null` если шаблон не

соответствует `input` строке. Строка результата извлекается из `MatchResult?` объекта `value` свойства. Параметр `startIndex` является необязательным с значением по умолчанию 0.

Чтобы извлечь первый действительный номер телефона из строки с контактными данными:

```
val phoneNumber :String? = Regex(pattern = """\d{3}-\d{3}-\d{4}""")
    .find(input = "phone: 123-456-7890, e..")?.value // phoneNumber: 123-456-7890
```

При отсутствии действительного номера телефона во `input` строке переменная `phoneNumber` будет равна `null` .

findAll (вход: CharSequence, startIndex: Int): Последовательность

Возвращает все совпадения из `input` строки, которая соответствует `pattern` регулярного выражения.

Чтобы распечатать все номера, разделенные пробелом, из текста с буквами и цифрами:

```
val matchedResults = Regex(pattern = """\d+""").findAll(input = "ab12cd34ef")
val result = StringBuilder()
for (matchedText in matchedResults) {
    result.append(matchedText.value + " ")
}

println(result) // => 12 34
```

Переменная `matchedResults` представляет собой последовательность с объектами `MatchResult` . С помощью строки `input` без цифр функция `findAll(..)` вернет пустую последовательность.

matchEntire (вход: CharSequence): MatchResult?

Если все символы `input` строки соответствуют `pattern` регулярного выражения, будет возвращена строка, равная `input` . Else, `null` будет возвращен.

Возвращает входную строку, если вся строка ввода - это число:

```
val a = Regex("""\d+""").matchEntire("100")?.value             // a: 100
val b = Regex("""\d+""").matchEntire("100 dollars")?.value     // b: null
```

match (input: CharSequence): Boolean

Возвращает true, если вся строка ввода соответствует шаблону регулярного выражения. В противном случае.

Тесты, если две строки содержат только цифры:

```
val regex = Regex(pattern = """\d+""")
regex.matches(input = "50")             // => true
regex.matches(input = "50 dollars")     // => false
```

containsMatchIn (вход: CharSequence): Boolean

Возвращает true, если часть входной строки соответствует шаблону регулярного выражения. В противном случае.

Проверьте, содержит ли две строки по крайней мере одну цифру:

```
Regex("""\d+""").containsMatchIn("50 dollars")       // => true
Regex("""\d+""").containsMatchIn("Fifty dollars")    // => false
```

split (вход: CharSequence, limit: Int): Список

Возвращает новый список без совпадений регулярных выражений.

Для возврата списков без цифр:

```
val a = Regex("""\d+""").split("ab12cd34ef")     // a: [ab, cd, ef]
val b = Regex("""\d+""").split("This is a test") // b: [This is a test]
```

В каждом списке есть один элемент списка. Первая строка `input` содержит три числа. Это приводит к списку с тремя элементами.

replace (input: CharSequence, replacement: String): String

Заменяет все совпадения `pattern` регулярного выражения в строке `input` с помощью строки замены.

Чтобы заменить все цифры в строке на x:

```
val result = Regex("""\d+""").replace("ab12cd34ef", "x") // result: abxcdxef
```

Прочитайте Regex онлайн: https://riptutorial.com/ru/kotlin/topic/8364/regex

глава 8: Аннотации

Examples

Объявление аннотации

Аннотации - это средства привязки метаданных к коду. Чтобы объявить аннотацию, добавьте модификатор аннотации перед классом:

```
annotation class Strippable
```

Аннотации могут иметь мета-анотации:

```
    @Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
    annotation class Strippable
```

Аннотации, как и другие классы, могут иметь конструкторы:

```
annotation class Strippable(val importanceValue: Int)
```

Но в отличие от других классов, они ограничены следующими типами:

- типы, которые соответствуют примитивным типам Java (Int, Long и т. д.);
- строки
- классы (Foo :: class)
- перечислений
- другие аннотации
- массивы типов, перечисленных выше

Мета-аннотаций

При объявлении аннотации метаинформация может быть включена с использованием следующих мета-аннотаций:

- `@Target` : указывает возможные типы элементов, которые могут быть аннотированы аннотацией (классы, функции, свойства, выражения и т. Д.).

- `@Retention` указывает, `@Retention` ли аннотация в скомпилированных файлах классов и отображается ли она через отражение во время выполнения (по умолчанию оба значения являются true).

- `@Repeatable` позволяет использовать одну и ту же аннотацию для одного элемента несколько раз.

- `@MustBeDocumented` указывает, что аннотация является частью общедоступного API и должна быть включена в подпись класса или метода, указанную в созданной документации API.

Пример:

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
annotation class Fancy
```

Прочитайте Аннотации онлайн: https://riptutorial.com/ru/kotlin/topic/4074/аннотации

глава 9: Базовый Лямбдас

Синтаксис

- Явные параметры:
- {parameterName: ParameterType, otherParameterName: OtherParameterType -> anExpression ()}
- Выведенные параметры:
- val add: (Int, Int) -> Int = {a, b -> a + b}
- Один параметр - `it` сокращенный
- val square: (Int) -> Int = {it * it}
- Подпись:
- () -> ResultType
- (InputType) -> ResultType
- (InputType1, InputType2) -> ResultType

замечания

Параметры типа ввода могут быть опущены, когда их можно оставить без внимания, когда они могут быть выведены из контекста. Например, скажем, что у вас есть функция класса, который выполняет функцию:

```
data class User(val fistName: String, val lastName: String) {
    fun username(userNameGenerator: (String, String) -> String) =
        userNameGenerator(firstName, secondName)
}
```

Вы можете использовать эту функцию, передав лямбда, и поскольку параметры уже указаны в сигнатуре функции, нет необходимости повторно объявлять их в выражении лямбда:

```
val user = User("foo", "bar")
println(user.userName { firstName, secondName ->
     "${firstName.toUppercase}"_"${secondName.toUppercase}"
 }) // prints FOO_BAR
```

Это также применяется, когда вы назначаете лямбда переменной:

```
//valid:
val addition: (Int, Int) = { a, b -> a + b }
//valid:
val addition = { a: Int, b: Int -> a + b }
//error (type inference failure):
val addition = { a, b -> a + b }
```

Когда лямбда принимает один параметр, и тип может быть выведен из контекста, вы можете обратиться к параметру с помощью `it` .

```
listOf(1,2,3).map { it * 2 } // [2,4,6]
```

Examples

Лямбда как параметр для функции фильтрации

```
val allowedUsers = users.filter { it.age > MINIMUM_AGE }
```

Лямбда передавалась как переменная

```
val isOfAllowedAge = { user: User -> user.age > MINIMUM_AGE }
val allowedUsers = users.filter(isOfAllowedAge)
```

Лямбда для сравнения вызова функции

Секундомер общего назначения для определения времени выполнения функции для запуска:

```
object Benchmark {
    fun realtime(body: () -> Unit): Duration {
        val start = Instant.now()
        try {
            body()
        } finally {
            val end = Instant.now()
            return Duration.between(start, end)
        }
    }
}
```

Использование:

```
val time = Benchmark.realtime({
    // some long-running code goes here ...
})
println("Executed the code in $time")
```

Прочитайте Базовый Лямбдас онлайн: https://riptutorial.com/ru/kotlin/topic/5878/базовый-лямбдас

глава 10: Варгарные параметры в функциях

Синтаксис

- **Vararg Ключевое слово** : `vararg` используется в объявлении метода, чтобы указать, что будет принято переменное число параметров.
- **Оператор распространения** : звездочка (`*`) перед массивом, который используется в вызовах функций, «разворачивает» содержимое в отдельные параметры.

Examples

Основы: использование ключевого слова vararg

Определите функцию, используя ключевое слово `vararg` .

```
fun printNumbers(vararg numbers: Int) {
    for (number in numbers) {
        println(number)
    }
}
```

Теперь вы можете передать столько параметров (нужного типа) в нужную вам функцию.

```
printNumbers(0, 1)                // Prints "0" "1"
printNumbers(10, 20, 30, 500)     // Prints "10" "20" "30" "500"
```

> **Примечания.** Параметры Vararg *должны* быть последним параметром в списке параметров.

Оператор распространения: передача массивов в функции vararg

Массивы могут быть переданы в функции vararg с использованием **оператора Spread** , `*` .

Предполагая, что существует следующая функция ...

```
fun printNumbers(vararg numbers: Int) {
    for (number in numbers) {
        println(number)
    }
}
```

Вы можете **передать массив** в функцию так ...

```
val numbers = intArrayOf(1, 2, 3)
printNumbers(*numbers)

// This is the same as passing in (1, 2, 3)
```

Оператор распространения также может использоваться **в середине** параметров ...

```
val numbers = intArrayOf(1, 2, 3)
printNumbers(10, 20, *numbers, 30, 40)

// This is the same as passing in (10, 20, 1, 2, 3, 30, 40)
```

Прочитайте Варгарные параметры в функциях онлайн: https://riptutorial.com/ru/kotlin/topic/5835/варгарные-параметры-в-функциях

глава 11: Делегирование класса

Вступление

Класс Kotlin может реализовать интерфейс, делегируя его методы и свойства другому объекту, реализующему этот интерфейс. Это обеспечивает способ создания поведения с использованием ассоциации, а не наследования.

Examples

Передача метода другому классу

```
interface Foo {
    fun example()
}

class Bar {
    fun example() {
        println("Hello, world!")
    }
}

class Baz(b : Bar) : Foo by b

Baz(Bar()).example()
```

Пример печатает `Hello, world!`

Прочитайте Делегирование класса онлайн: https://riptutorial.com/ru/kotlin/topic/10575/делегирование-класса

глава 12: Делегированные свойства

Вступление

Kotlin может делегировать реализацию свойства объекту обработчика. Некоторые стандартные обработчики включены, например, ленивая инициализация или наблюдаемые свойства. Также могут быть созданы пользовательские обработчики.

Examples

Ленивая инициализация

```
val foo : Int by lazy { 1 + 1 }
println(foo)
```

Пример печатает `2` .

Наблюдаемые свойства

```
var foo : Int by Delegates.observable("1") { property, oldValue, newValue ->
    println("${property.name} was changed from $oldValue to $newValue")
}
foo = 2
```

Пример печати `foo was changed from 1 to 2`

Свойства карты

```
val map = mapOf("foo" to 1)
val foo : String by map
println(foo)
```

Пример: `1`

Пользовательское

```
class MyDelegate {
    operator fun getValue(owner: Any?, property: KProperty<*>): String {
        return "Delegated value"
    }
}

val foo : String by MyDelegate()
println(foo)
```

Пример отображает `Delegated value`

Делегат Может использоваться как слой для уменьшения шаблона

Рассмотрим систему Null Type Kotlin и `WeakReference<T>` .

Итак, скажем, нам нужно сохранить какую-то ссылку, и мы хотели избежать утечек памяти, здесь `WeakReference` .

возьмите, например, следующее:

```
class MyMemoryExpensiveClass {
    companion object {
        var reference: WeakReference<MyMemoryExpensiveClass>? = null

        fun doWithReference(block: (MyMemoryExpensiveClass) -> Unit) {
            reference?.let {
                it.get()?.let(block)
            }
        }
    }

    init {
        reference = WeakReference(this)
    }
}
```

Теперь это только с одним WeakReference. Чтобы уменьшить этот шаблон, мы можем использовать персонализированный делегат свойств, чтобы помочь нам в этом:

```
class WeakReferenceDelegate<T>(initialValue: T? = null) : ReadWriteProperty<Any, T?> {
    var reference = WeakReference(initialValue)
        private set

    override fun getValue(thisRef: Any, property: KProperty<*>): T? = reference.get()

    override fun setValue(thisRef: Any, property: KProperty<*>, value: T?) {
        reference = WeakReference(value)
    }
}
```

Итак, теперь мы можем использовать переменные, которые обернуты с помощью `WeakReference` как обычные переменные с `WeakReference` значением!

```
class MyMemoryExpensiveClass {
    companion object {
        var reference: MyMemoryExpensiveClass? by
WeakReferenceDelegate<MyMemoryExpensiveClass>()

        fun doWithReference(block: (MyMemoryExpensiveClass) -> Unit) {
            reference?.let(block)
        }
    }
```

```
    init {
        reference = this
    }
}
```

Прочитайте Делегированные свойства онлайн: https://riptutorial.com/ru/kotlin/topic/10571/делегированные-свойства

глава 13: Дженерики

Вступление

Список может содержать числа, слова или действительно что угодно. Вот почему мы называем *общий* список.

Генерики обычно используются для определения типов, которые может содержать класс, и типа, который в данный момент хранит объект.

Синтаксис

- class *ClassName* < ***TypeName*** >
- class *ClassName* <*>
- *ClassName* <in ***UpperBound*** >
- *ClassName* <out ***LowerBound*** >
- class *Name* < ***TypeName*** : ***UpperBound*** >

параметры

параметр	подробности
TypeName	Тип Имя общего параметра
Верхняя граница	Ковариантный тип
Нижняя граница	Контравариантный тип
ИмяКласса	Название класса

замечания

Подразумеваемая верхняя граница - Nullable

В Kotlin Generics верхняя граница параметра типа `T` будет `Any?` , Поэтому для этого класса:

```
class Consumer<T>
```

Параметр типа `T` действительно `T: Any?` , Чтобы сделать верхнюю границу с

недействительными значениями, явно определенную `T: Any` . Например:

```
class Consumer<T: Any>
```

Examples

Отклонение на сайте

Отклонение на уровне объявления можно рассматривать как объявление разницы в использовании сайта один раз и на всех сайтах-участницах.

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

fun charSequencesConsumer() : Consumer<CharSequence>() = ...

val stringConsumer : Consumer<String> = charSequenceConsumer() // OK since in-projection
val anyConsumer : Consumer<Any> = charSequenceConsumer() // Error, Any cannot be passed

val outConsumer : Consumer<out CharSequence> = ... // Error, T is `in`-parameter
```

Широко распространенными примерами дисперсии объявления-сайта являются `List<out T>` , который является неизменным, поэтому `T` появляется только как тип возвращаемого значения и `Comparator<in T>` , который принимает только `T` как аргумент.

Распределение на месте

Разница между сайтом и сайтом похожа на маскировку Java:

Из-проекции:

```
val takeList : MutableList<out SomeType> = ... // Java: List<? extends SomeType>

val takenValue : SomeType = takeList[0] // OK, since upper bound is SomeType

takeList.add(takenValue) // Error, lower bound for generic is not specified
```

В-проекции:

```
val putList : MutableList<in SomeType> = ... // Java: List<? super SomeType>

val valueToPut : SomeType = ...
putList.add(valueToPut) // OK, since lower bound is SomeType

putList[0] // This expression has type Any, since no upper bound is specified
```

Звезда-проекции

```
val starList : MutableList<*> = ... // Java: List<?>
```

```
starList[0] // This expression has type Any, since no upper bound is specified
starList.add(someValue) // Error, lower bound for generic is not specified
```

Смотрите также:

- Возможность взаимодействия разновидностей Generic при вызове Kotlin с Java.

Прочитайте Дженерики онлайн: https://riptutorial.com/ru/kotlin/topic/1147/дженерики

глава 14: Идиомы

Examples

Создание DTO (POJO / POCOs)

Классы данных в kotlin - это классы, созданные только для хранения данных. Такие классы отмечены как `data` :

```
data class User(var firstname: String, var lastname: String, var age: Int)
```

В приведенном выше коде создается класс `User` со следующими автоматически генерируемыми:

- Getters и Setters для всех свойств (getters только для `val` s)
- `equals()`
- `hashcode()`
- `toString()`
- `copy()`
- `componentN()` (где `N` - соответствующее свойство в порядке объявления)

Так же, как и для функции, значения по умолчанию также могут быть указаны:

```
data class User(var firstname: String = "Joe", var lastname: String = "Bloggs", var age: Int =
20)
```

Более подробную информацию можно найти здесь « Классы данных» .

Фильтрация списка

```
val list = listOf(1,2,3,4,5,6)

//filter out even numbers

val even = list.filter { it % 2 == 0 }

println(even) //returns [2,4]
```

Делегировать в класс, не предоставляя его в публичном конструкторе

Предположим, вы хотите делегировать класс, но вы не хотите предоставлять делегированный класс в параметре конструктора. Вместо этого вы хотите создать его конфиденциально, заставив вызывающего абонента конструктора не знать об этом. Сначала это может показаться невозможным, поскольку делегирование классов позволяет делегировать только параметры конструктора. Однако есть способ сделать это, как

указано в этом ответе :

```
class MyTable private constructor(table: Table<Int, Int, Int>) : Table<Int, Int, Int> by table
{

    constructor() : this(TreeBasedTable.create()) // or a different type of table if desired

}
```

С этим вы можете просто вызвать конструктор `MyTable` следующим образом: `MyTable()` . `Table<Int, Int, Int>` с которой делегаты `MyTable` будут создаваться конфиденциально. Консоль конструктора ничего не знает об этом.

Этот пример основан на этом вопросе SO .

Сериализуемый и serialVersionUid в Котлине

Чтобы создать `serialVersionUID` для класса в Kotlin, у вас есть несколько вариантов, связанных с добавлением члена к сопутствующему объекту класса.

Самый сжатый байт-код исходит из `private const val` который станет частной статической переменной в содержащем классе, в этом случае `MySpecialCase` :

```
class MySpecialCase : Serializable {
    companion object {
        private const val serialVersionUID: Long = 123
    }
}
```

Вы также можете использовать эти формы, **каждый из которых имеет побочный эффект от методов getter / setter,** которые не нужны для сериализации ...

```
class MySpecialCase : Serializable {
    companion object {
        private val serialVersionUID: Long = 123
    }
}
```

Это создает статическое поле, но также создает getter, а также `getSerialVersionUID` на сопутствующем объекте, который не нужен.

```
class MySpecialCase : Serializable {
    companion object {
        @JvmStatic private val serialVersionUID: Long = 123
    }
}
```

Это создает статическое поле, но также создает статический getter, а `getSerialVersionUID` в содержащем классе `MySpecialCase` который не нужен.

Но все они работают как метод добавления `serialVersionUID` в класс `Serializable` .

Свободные методы в Котлине

Свободные методы в Котлине могут быть такими же, как Java:

```
fun doSomething() {
   someOtherAction()
   return this
}
```

Но вы также можете сделать их более функциональными, создав функцию расширения, такую как:

```
fun <T: Any> T.fluently(func: ()->Unit): T {
    func()
    return this
}
```

Которая затем позволяет более явно бегло выполнять функции:

```
fun doSomething() {
   return fluently { someOtherAction() }
}
```

Используйте let или также для упрощения работы с объектами с нулевым значением

`let` in Kotlin создает локальную привязку с объекта, на который он был вызван. Пример:

```
val str = "foo"
str.let {
    println(it) // it
}
```

Это напечатает `"foo"` и вернет `Unit` .

Разница между `let` и `also` заключается в том, что вы можете вернуть любое значение из блока `let` . `also` в другой руке всегда будет единица `Unit` .

Теперь почему это полезно, спросите вы? Потому что, если вы вызываете метод, который может возвращать значение `null` и вы хотите запустить некоторый код только тогда, когда это возвращаемое значение не является `null` вы можете использовать `let` или `also` как это:

```
val str: String? = someFun()
str?.let {
    println(it)
}
```

Эта часть кода будет запускать только блок `let` когда `str` не является `null` . Обратите внимание на `null` оператор безопасности (`?`).

Использование применяется для инициализации объектов или для достижения цепочки методов

В документации по `apply` говорится следующее:

> вызывает указанный функциональный блок с `this` значением в качестве приемника и возвращает `this` значение.

В то время как kdoc не так полезно `apply` это действительно полезная функция. При `apply` условий неспециалиста устанавливает область, в которой `this` связано с объектом, на который вы `apply` . Это позволяет вам сэкономить некоторый код, когда вам нужно вызвать несколько методов для объекта, который вы затем вернете позже. Пример:

```
File(dir).apply { mkdirs() }
```

Это то же самое, что написать это:

```
fun makeDir(String path): File {
    val result = new File(path)
    result.mkdirs()
    return result
}
```

Прочитайте Идиомы онлайн: https://riptutorial.com/ru/kotlin/topic/2273/идиомы

глава 15: Изменяется

Вступление

Выражения диапазона формируются с помощью функций rangeTo, которые имеют операторную форму, которая дополняется in и! In. Диапазон определен для любого сопоставимого типа, но для интегральных примитивных типов он имеет оптимизированную реализацию

Examples

Интегральные диапазоны типов

Диапазоны интегрального типа (IntRange, LongRange, CharRange) имеют дополнительную функцию: их можно повторить. Компилятор позаботится о преобразовании этого аналогично в индексированный для Java цикл, без дополнительных накладных расходов

```
for (i in 1..4) print(i) // prints "1234"
for (i in 4..1) print(i) // prints nothing
```

Функция downTo ()

если вы хотите перебрать числа в обратном порядке? Это просто. Вы можете использовать функцию downTo (), определенную в стандартной библиотеке

```
for (i in 4 downTo 1) print(i) // prints "4321"
```

step ()

Можно ли перебирать числа с произвольным шагом, не равным 1? Конечно, функция step () поможет вам

```
for (i in 1..4 step 2) print(i) // prints "13"
for (i in 4 downTo 1 step 2) print(i) // prints "42"
```

пока функция

Чтобы создать диапазон, который не включает его конечный элемент, вы можете использовать функцию until:

```
for (i in 1 until 10) { // i in [1, 10), 10 is excluded
println(i)
}
```

Прочитайте Изменяется онлайн: https://riptutorial.com/ru/kotlin/topic/10121/изменяется

глава 16: Интерфейсы

замечания

См. Также: Справочная документация Kotlin для интерфейсов: Интерфейсы

Examples

Основной интерфейс

Интерфейс Kotlin содержит декларации абстрактных методов и реализации метода по умолчанию, хотя они не могут сохранять состояние.

```
interface MyInterface {
    fun bar()
}
```

Теперь этот интерфейс может быть реализован классом следующим образом:

```
class Child : MyInterface {
   override fun bar() {
       print("bar() was called")
   }
}
```

Интерфейс с реализациями по умолчанию

Интерфейс в Kotlin может иметь стандартные реализации для функций:

```
interface MyInterface {
    fun withImplementation() {
      print("withImplementation() was called")
    }
}
```

Классы, реализующие такие интерфейсы, смогут использовать эти функции без переопределения

```
class MyClass: MyInterface {
    // No need to reimplement here
}
val instance = MyClass()
instance.withImplementation()
```

свойства

Реализации по умолчанию также работают для получателей и сеттеров:

```
interface MyInterface2 {
    val helloWorld
        get() = "Hello World!"
}
```

Реализации интерфейсов не могут использовать поддерживающие поля

```
interface MyInterface3 {
    // this property won't compile!
    var helloWorld: Int
        get() = field
        set(value) { field = value }
}
```

Несколько реализаций

Когда несколько интерфейсов реализуют одну и ту же функцию или все из них определяют с одним или несколькими реализациями, производный класс должен вручную разрешить правильный вызов

```
interface A {
    fun notImplemented()
    fun implementedOnlyInA() { print("only A") }
    fun implementedInBoth() { print("both, A") }
    fun implementedInOne() { print("implemented in A") }
}

interface B {
    fun implementedInBoth() { print("both, B") }
    fun implementedInOne() // only defined
}

class MyClass: A, B {
    override fun notImplemented() { print("Normal implementation") }

    // implementedOnlyInA() can by normally used in instances

    // class needs to define how to use interface functions
    override fun implementedInBoth() {
        super<B>.implementedInBoth()
        super<A>.implementedInBoth()
    }

    // even if there's only one implementation, there multiple definitions
    override fun implementedInOne() {
        super<A>.implementedInOne()
        print("implementedInOne class implementation")
    }
}
```

Свойства в интерфейсах

Вы можете объявлять свойства в интерфейсах. Так как интерфейс не может иметь состояние, вы можете объявить свойство только абстрактным или предоставить реализацию по умолчанию для аксессуаров.

```
interface MyInterface {
    val property: Int // abstract

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(property)
    }
}

class Child : MyInterface {
    override val property: Int = 29
}
```

Конфликты при реализации нескольких интерфейсов с реализацией по умолчанию

При реализации более одного интерфейса, который имеет методы с тем же именем, которые включают реализации по умолчанию, он неоднозначен для компилятора, реализация которого должна использоваться. В случае конфликта разработчик должен переопределить конфликтующий метод и предоставить пользовательскую реализацию. Эта реализация может выбрать делегирование реализации по умолчанию или нет.

```
interface FirstTrait {
    fun foo() { print("first") }
    fun bar()
}

interface SecondTrait {
    fun foo() { print("second") }
    fun bar() { print("bar") }
}

class ClassWithConflict : FirstTrait, SecondTrait {
    override fun foo() {
        super<FirstTrait>.foo()  // delegate to the default implementation of FirstTrait
        super<SecondTrait>.foo() // delegate to the default implementation of SecondTrait
    }

    // function bar() only has a default implementation in one interface and therefore is ok.
}
```

супер ключевое слово

```
interface MyInterface {
    fun funcOne() {
        //optional body
        print("Function with default implementation")
```

```
    }
}
```

Если метод интерфейса имеет свою собственную реализацию по умолчанию, мы можем использовать ключевое слово super для доступа к нему.

```
super.funcOne()
```

Прочитайте Интерфейсы онлайн: https://riptutorial.com/ru/kotlin/topic/900/интерфейсы

глава 17: Исключения

Examples

Исключение исключения с помощью try-catch-finally

Захватывающие исключения в Kotlin очень похожи на Java

```
try {
    doSomething()
}
catch(e: MyException) {
    handle(e)
}
finally {
    cleanup()
}
```

Вы также можете поймать несколько исключений

```
try {
    doSomething()
}
catch(e: FileSystemException) {
    handle(e)
}
catch(e: NetworkException) {
    handle(e)
}
catch(e: MemoryException) {
    handle(e)
}
finally {
    cleanup()
}
```

`try` также является выражением и может возвращать значение

```
val s: String? = try { getString() } catch (e: Exception) { null }
```

Котлин не проверял исключения, поэтому вам не нужно ломать никаких исключений.

```
fun fileToString(file: File) : String {
    //readAllBytes throws IOException, but we can omit catching it
    fileContent = Files.readAllBytes(file)
    return String(fileContent)
}
```

Прочитайте Исключения онлайн: https://riptutorial.com/ru/kotlin/topic/7246/исключения

глава 18: Коллекции

Вступление

В отличие от многих языков, Котлин различает изменчивые и неизменные коллекции (списки, наборы, карты и т. Д.). Точный контроль над тем, когда коллекции можно редактировать, полезно для устранения ошибок и для разработки хороших API.

Синтаксис

- listOf, mapOf и setOf возвращают объекты только для чтения, которые нельзя добавлять или удалять.
- Если вы хотите добавить или удалить элементы, вы должны использовать arrayListOf, hashMapOf, hashSetOf, linkedMapOf (LinkedHashMap), linkedSetOf (LinkedHashSet), mutableListOf (коллекция Kotlin MultableList), mutableMapOf (коллекция Kotlin MultableMap), mutableSetOf (коллекция Kotlin MultableSet), sortedMapOf или sortedSetOf
- Каждая коллекция имеет такие методы, как first (), last (), get () и лямбда-функции, такие как фильтр, карта, объединение, сокращение и многие другие.

Examples

Использование списка

```
// Create a new read-only List<String>
val list = listOf("Item 1", "Item 2", "Item 3")
println(list) // prints "[Item 1, Item 2, Item 3]"
```

Использование карты

```
// Create a new read-only Map<Integer, String>
val map = mapOf(Pair(1, "Item 1"), Pair(2, "Item 2"), Pair(3, "Item 3"))
println(map) // prints "{1=Item 1, 2=Item 2, 3=Item 3}"
```

Использование набора

```
// Create a new read-only Set<String>
val set = setOf(1, 3, 5)
println(set) // prints "[1, 3, 5]"
```

Прочитайте Коллекции онлайн: https://riptutorial.com/ru/kotlin/topic/8846/коллекции

глава 19: Массивы

Examples

Общие массивы

Общие массивы в Котлине представлены `Array<T>` .

Чтобы создать пустой массив, используйте `emptyArray<T>()` :

```
val empty = emptyArray<String>()
```

Чтобы создать массив с заданным размером и начальными значениями, используйте конструктор:

```
var strings = Array<String>(size = 5, init = { index -> "Item #$index" })
print(Arrays.toString(a)) // prints "[Item #0, Item #1, Item #2, Item #3, Item #4]"
print(a.size) // prints 5
```

Массивы имеют функции `get(index: Int): T` и `set(index: Int, value: T)` :

```
strings.set(2, "ChangedItem")
print(strings.get(2)) // prints "ChangedItem"

// You can use subscription as well:
strings[2] = "ChangedItem"
print(strings[2]) // prints "ChangedItem"
```

Массивы примитивов

Эти типы **не** наследуют от `Array<T>` чтобы избежать бокса, однако они имеют одинаковые атрибуты и методы.

Тип Котлина	Заводская функция	Тип JVM
`BooleanArray`	`booleanArrayOf(true, false)`	`boolean[]`
`ByteArray`	`byteArrayOf(1, 2, 3)`	`byte[]`
`CharArray`	`charArrayOf('a', 'b', 'c')`	`char[]`
`DoubleArray`	`doubleArrayOf(1.2, 5.0)`	`double[]`
`FloatArray`	`floatArrayOf(1.2, 5.0)`	`float[]`
`IntArray`	`intArrayOf(1, 2, 3)`	`int[]`
`LongArray`	`longArrayOf(1, 2, 3)`	`long[]`

Тип Котлина	Заводская функция	Тип JVM
ShortArray	shortArrayOf(1, 2, 3)	short[]

расширения

`average()` определяется для `Byte` , `Int` , `Long` , `Short` , `Double` , `Float` и всегда возвращает `Double` :

```
val doubles = doubleArrayOf(1.5, 3.0)
print(doubles.average()) // prints 2.25

val ints = intArrayOf(1, 4)
println(ints.average()) // prints 2.5
```

`component1()` , `component2()` , ... `component5()` возвращает элемент массива

`getOrNull(index: Int)` возвращает значение null, если индекс за пределами границ, иначе элемент массива

`first()` , `last()`

`toHashSet()` возвращает `HashSet<T>` всех элементов

`sortedArray()` , `sortedArrayDescending()` создает и возвращает новый массив с отсортированными элементами текущего

`sort()` , `sortDescending` сортировать массив на месте

`min()` , `max()`

Итерационный массив

Вы можете распечатать элементы массива , используя цикл такого же , как расширенный цикл Java, но вам нужно изменить ключевое слово от `:` до `in` .

```
val asc = Array(5, { i -> (i * i).toString() })
for(s : String in asc){
    println(s);
}
```

Вы также можете изменить тип данных для цикла.

```
val asc = Array(5, { i -> (i * i).toString() })
for(s in asc){
    println(s);
}
```

Создать массив

```
val a = arrayOf(1, 2, 3) // creates an Array<Int> of size 3 containing [1, 2, 3].
```

Создание массива с использованием закрытия

```
val a = Array(3) { i -> i * 2 } // creates an Array<Int> of size 3 containing [0, 2, 4]
```

Создать неинициализированный массив

```
val a = arrayOfNulls<Int>(3) // creates an Array<Int?> of [null, null, null]
```

Возвращаемый массив всегда будет иметь тип с нулевым значением. Массивы элементов, не подлежащих обнулению, не могут быть созданы неинициализированными.

Прочитайте Массивы онлайн: https://riptutorial.com/ru/kotlin/topic/5722/массивы

глава 20: Методы расширения

Синтаксис

- fun ТипName.extensionName (params, ...) {/ * body * /} // Объявление
- fun <T: Any> TypeNameWithGenerics <T> .extensionName (params, ...) {/ * body * /} // Объявление с помощью Generics
- myObj.extensionName (args, ...) // вызов

замечания

Расширения разрешаются **статически** . Это означает, что используемый метод расширения определяется ссылочным типом переменной, к которой вы обращаетесь; неважно, какой тип переменной находится во время выполнения, всегда будет вызываться тот же метод расширения. Это связано с тем, что **объявление метода расширения фактически не добавляет член к типу приемника** .

Examples

Расширения верхнего уровня

Методы расширения верхнего уровня не содержатся в классе.

```
fun IntArray.addTo(dest: IntArray) {
    for (i in 0 .. size - 1) {
        dest[i] += this[i]
    }
}
```

Выше метода расширения определен для типа `IntArray` . Обратите внимание, что объект, для которого определен метод расширения (называемый **получателем**), доступен с использованием ключевого слова `this` .

Это расширение можно вызвать так:

```
val myArray = intArrayOf(1, 2, 3)
intArrayOf(4, 5, 6).addTo(myArray)
```

Потенциальная Pitfall: расширения разрешаются статически

Вызываемый метод расширения определяется во время компиляции на основе ссылочного типа доступной переменной. Неважно, какой тип переменной находится во время выполнения, всегда будет вызываться тот же метод расширения.

```
open class Super

class Sub : Super()

fun Super.myExtension() = "Defined for Super"

fun Sub.myExtension() = "Defined for Sub"

fun callMyExtension(myVar: Super) {
    println(myVar.myExtension())
}

callMyExtension(Sub())
```

В приведенном выше примере будет напечатан `"Defined for Super"` , потому что объявленный тип переменной `myVar` - `Super` .

Образец, простирающийся долго, чтобы отобразить читаемую пользователем строку

Для любого значения типа `Int` или `Long` для визуализации строки, читаемой человеком:

```
fun Long.humanReadable(): String {
    if (this <= 0) return "0"
    val units = arrayOf("B", "KB", "MB", "GB", "TB", "EB")
    val digitGroups = (Math.log10(this.toDouble())/Math.log10(1024.0)).toInt();
    return DecimalFormat("#,##0.#").format(this/Math.pow(1024.0, digitGroups.toDouble())) + "
" + units[digitGroups];
}

fun Int.humanReadable(): String {
    return this.toLong().humanReadable()
}
```

Затем легко использовать как:

```
println(1999549L.humanReadable())
println(someInt.humanReadable())
```

Пример расширения Java 7+ Path class

Общим вариантом использования методов расширения является улучшение существующего API. Вот примеры добавления `exist` , `notExists` и `deleteRecursively` к классу Java 7+ `Path` :

```
fun Path.exists(): Boolean = Files.exists(this)
fun Path.notExists(): Boolean = !this.exists()
fun Path.deleteRecursively(): Boolean = this.toFile().deleteRecursively()
```

Который теперь можно вызвать в этом примере:

```
val dir = Paths.get(dirName)
```

```
if (dir.exists()) dir.deleteRecursively()
```

Использование функций расширения для повышения удобочитаемости

В Котлине вы можете написать код вроде:

```
val x: Path = Paths.get("dirName").apply {
    if (Files.notExists(this)) throw IllegalStateException("The important file does not
exist")
}
```

Но использование `apply` не совсем ясно в отношении ваших намерений. Иногда более понятно создать аналогичную функцию расширения, чтобы фактически переименовать действие и сделать его более очевидным. Это не должно выходить из-под контроля, но для очень распространенных действий, таких как проверка:

```
infix inline fun <T> T.verifiedBy(verifyWith: (T) -> Unit): T {
    verifyWith(this)
    return this
}

infix inline fun <T: Any> T.verifiedWith(verifyWith: T.() -> Unit): T {
    this.verifyWith()
    return this
}
```

Теперь вы можете написать код как:

```
val x: Path = Paths.get("dirName") verifiedWith {
    if (Files.notExists(this)) throw IllegalStateException("The important file does not
exist")
}
```

Что теперь даст людям знать, чего ожидать в пределах параметра лямбда.

Обратите внимание, что параметр типа `T` для `verifiedBy` такой же, как `T: Any?` что даже типы с нулевым значением смогут использовать эту версию расширения. Хотя `verifiedWith` требует не-nullable.

Пример расширения Java 8 Временные классы для отображения строки в формате ISO

С этим заявлением:

```
fun Temporal.toIsoString(): String = DateTimeFormatter.ISO_INSTANT.format(this)
```

Теперь вы можете просто:

```
val dateAsString = someInstant.toIsoString()
```

Функции расширения для объектов-компаньонов (появление статических функций)

Если вы хотите расширить класс так: если вы статическая функция, например, для класса `Something` добавьте статическую функцию `fromString` , это может работать, только если класс имеет сопутствующий объект и что функция расширения объявлена на сопутствующем объекте :

```
class Something {
    companion object {}
}

class SomethingElse {
}

fun Something.Companion.fromString(s: String): Something = ...

fun SomethingElse.fromString(s: String): SomethingElse = ...

fun main(args: Array<String>) {
    Something.fromString("") //valid as extension function declared upon the
                             //companion object

    SomethingElse().fromString("") //valid, function invoked on instance not
                                   //statically

    SomethingElse.fromString("") //invalid
}
```

Ложное расширение

Предположим, вы хотите создать свойство расширения, которое дорого вычислить. Таким образом, вы хотели бы кэшировать вычисление, используя делегат lazy property и ссылаться на текущий экземпляр (`this`), но вы не можете этого сделать, как объяснено в Kotlin выпусках KT-9686 и KT-13053 . Тем не менее, существует официальное обходное решение, представленное здесь .

В этом примере свойство расширения является `color` . Он использует явный `colorCache` который может быть использован с `this` поскольку нет `lazy` :

```
class KColor(val value: Int)

private val colorCache = mutableMapOf<KColor, Color>()

val KColor.color: Color
    get() = colorCache.getOrPut(this) { Color(value, true) }
```

Расширения для упрощения ссылки Вид из кода

Вы можете использовать расширения для ссылочного вида, без шаблонов после создания представлений.

Оригинальная идея - библиотека Anko

расширения

```
inline fun <reified T : View> View.find(id: Int): T = findViewById(id) as T
inline fun <reified T : View> Activity.find(id: Int): T = findViewById(id) as T
inline fun <reified T : View> Fragment.find(id: Int): T = view?.findViewById(id) as T
inline fun <reified T : View> RecyclerView.ViewHolder.find(id: Int): T =
itemView?.findViewById(id) as T

inline fun <reified T : View> View.findOptional(id: Int): T? = findViewById(id) as? T
inline fun <reified T : View> Activity.findOptional(id: Int): T? = findViewById(id) as? T
inline fun <reified T : View> Fragment.findOptional(id: Int): T? = view?.findViewById(id) as?
T
inline fun <reified T : View> RecyclerView.ViewHolder.findOptional(id: Int): T? =
itemView?.findViewById(id) as? T
```

использование

```
val yourButton by lazy { find<Button>(R.id.yourButtonId) }
val yourText by lazy { find<TextView>(R.id.yourTextId) }
val yourEdittextOptional by lazy { findOptional<EditText>(R.id.yourOptionEdittextId) }
```

Прочитайте Методы расширения онлайн: https://riptutorial.com/ru/kotlin/topic/613/методы-расширения

глава 21: Модификаторы видимости

Вступление

В Котлине существует 4 типа модификаторов видимости.

Публикация: Доступ к ней возможен из любого места.

Частный: доступ к этому может быть получен только из кода модуля.

Protected: Доступ к этому может быть доступен только из класса, определяющего его и любых производных классов.

Внутренний: доступ к нему возможен только из области определения класса.

Синтаксис

- `<visibility modifier> val/var <variable name> = <value>`

Examples

Образец кода

Public: `public val name = "Avijit"`

Частное `private val name = "Avijit"` **:** `private val name = "Avijit"`

Protected: `protected val name = "Avijit"`

Внутренний: `internal val name = "Avijit"`

Прочитайте Модификаторы видимости онлайн: https://riptutorial.com/ru/kotlin/topic/10019/модификаторы-видимости

глава 22: Наследование класса

Вступление

Любой объектно-ориентированный язык программирования имеет некоторую форму наследования классов. Позвольте мне пересмотреть:

Представьте себе, что вам пришлось запрограммировать кучу фруктов: `Apples` , `Oranges` и `Pears` . Все они различаются по размеру, форме и цвету, поэтому у нас разные классы.

Но скажем, их различия не имеют значения на секунду, и вы просто хотите, чтобы `Fruit` , независимо от того, что именно? Какой тип возврата будет `getFruit()` ?

Ответ - класс `Fruit` . Мы создаем новый класс и накладываем на него все плоды!

Синтаксис

- открыть {базовый класс}
- class {Производный класс}: {Базовый класс} ({Аргументы инициализации})
- override {Function Definition}
- {DC-Object} - {Base Class} == true

параметры

параметр	подробности
Базовый класс	Класс, унаследованный от
Производный класс	Класс, который наследуется от базового класса
Инициировать аргументы	Аргументы, переданные конструктору базового класса
Определение функции	Функция в производном классе, которая отличается от кода в базовом классе
DC-Object	Объект «Производный класс-объект», который имеет тип производного класса

Examples

Основы: ключевое слово 'open'

В Котлине классы **по умолчанию окончательны,** что означает, что они не могут быть унаследованы.

Чтобы разрешить наследование для класса, используйте ключевое слово `open` .

```
open class Thing {
    // I can now be extended!
}
```

> **Примечание.** Абстрактные классы, закрытые классы и интерфейсы будут `open` по умолчанию.

Наследование полей из класса

Определение базового класса:

```
open class BaseClass {
    val x = 10
}
```

Определение производного класса:

```
class DerivedClass: BaseClass() {
    fun foo() {
        println("x is equal to " + x)
    }
}
```

Использование подкласса:

```
fun main(args: Array<String>) {
    val derivedClass = DerivedClass()
    derivedClass.foo() // prints: 'x is equal to 10'
}
```

Наследование методов из класса

Определение базового класса:

```
open class Person {
    fun jump() {
        println("Jumping...")
    }
}
```

Определение производного класса:

```
class Ninja: Person() {
    fun sneak() {
        println("Sneaking around...")
    }
}
```

Ниндзя имеет доступ ко всем методам в Person

```
fun main(args: Array<String>) {
    val ninja = Ninja()
    ninja.jump()  // prints: 'Jumping...'
    ninja.sneak() // prints: 'Sneaking around...'
}
```

Переопределение свойств и методов

Переопределение свойств (как для чтения, так и для изменяемых):

```
abstract class Car {
    abstract val name: String;
    open var speed: Int = 0;
}

class BrokenCar(override val name: String) : Car() {
    override var speed: Int
        get() = 0
        set(value) {
            throw UnsupportedOperationException("The car is bloken")
        }
}

fun main(args: Array<String>) {
    val car: Car = BrokenCar("Lada")
    car.speed = 10
}
```

Переопределение методов:

```
interface Ship {
    fun sail()
    fun sink()
}

object Titanic : Ship {

    var canSail = true

    override fun sail() {
        sink()
    }

```

```
    override fun sink() {
        canSail = false
    }
}
```

Прочитайте Наследование класса онлайн: https://riptutorial.com/ru/kotlin/topic/5622/наследование-класса

глава 23: Настройка сборки Kotlin

Examples

Конфигурация гребенки

`kotlin-gradle-plugin` используется для компиляции кода Kotlin с Gradle. В принципе, его версия должна соответствовать версии Kotlin, которую вы хотите использовать. Например, если вы хотите использовать Kotlin `1.0.3` , вам также понадобится aplly `kotlin-gradle-plugin` version `1.0.3` .

Это хорошая идея для экстернализации этой версии в `gradle.properties` или в `ExtraPropertiesExtension` :

```
buildscript {
   ext.kotlin_version = '1.0.3'

   repositories {
     mavenCentral()
   }

   dependencies {
     classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
   }
}
```

Затем вам необходимо применить этот плагин к вашему проекту. Способ, которым вы это делаете, отличается при ориентации на разные платформы:

Ориентация JVM

```
apply plugin: 'kotlin'
```

Ориентация на Android

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

Ориентация на JS

```
apply plugin: 'kotlin2js'
```

Это путь по умолчанию:

- kotlin источники: `src/main/kotlin`

- java-источники: `src/main/java`
- тесты kotlin: `src/test/kotlin`
- java-тесты: `src/test/java`
- ресурсы времени выполнения: `src/main/resources`
- тестовые ресурсы: `src/test/resources`

Возможно, вам придется настроить `SourceSets` если вы используете настраиваемый макет проекта.

Наконец, вам нужно будет добавить зависимость стандартной библиотеки Kotlin к вашему проекту:

```
dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

Если вы хотите использовать Kotlin Reflection, вам также нужно добавить `compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"`

Использование Android Studio

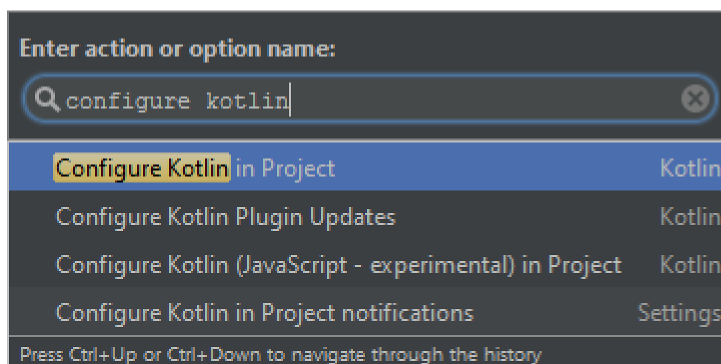
Android Studio может автоматически настроить Kotlin в проекте Android.

Установите плагин

Чтобы установить плагин Kotlin, откройте «Файл»> «Настройки»> «Редактор»> «Плагины»> «Установить плагин JetBrains ...> Kotlin> Install», затем перезапустите Android Studio при появлении запроса.

Настроить проект

Создайте проект Android Studio как обычно, затем нажмите `Ctrl` + `Shift` + `A`. В поле поиска введите «Настроить Kotlin в проекте» и нажмите «Ввод».



Android Studio изменит ваши файлы Gradle, чтобы добавить все необходимые зависимости.

Преобразование Java

Чтобы преобразовать ваши файлы Java в файлы Kotlin, нажмите `Ctrl` + `Shift` + `A` и найдите «Преобразовать файл Java в файл Kotlin». Это изменит расширение текущего файла на `.kt` и преобразует код в Kotlin.

```
package com.orangeflash81.myapplication;

public class Foo {
    private String name = "Joe Bloggs";

    String getName() { return name; }

    void setName(String value) { name = value; }
}
```

Миграция с Gradle с использованием скрипта Groovy на скрипт Kotlin

шаги:

- клонировать проект gradle-script-kotlin
- копировать / вставлять из клонированного проекта в ваш проект:
  - `build.gradle.kts`
  - `gradlew`
  - `gradlew.bat`
  - `settings.gradle`

- обновите содержимое `build.gradle.kts` на основе ваших потребностей, вы можете использовать в качестве вдохновения скрипты в проекте, просто клонированные или в одном из своих образцов

- теперь откройте Intellij и откройте свой проект в окне проводника, его следует признать проектом Gradle, если нет, сначала его раскройте.

- после открытия, пусть Intellij работает, откройте `build.gradle.kts` и проверьте, есть ли какая-либо ошибка. Если подсветка не работает и / или все отмечено красным цветом, затем закройте и снова запустите Intellij

- откройте окно Gradle и обновите его

Если вы находитесь в Windows, вы можете столкнуться с этой ошибкой , загрузить полный дистрибутив Gradle 3.3 и использовать его вместо этого. Связанный .

OSX и Ubuntu работают из коробки.

Небольшой бонус, если вы хотите избежать всех неприятностей публики на Maven и тому подобное, используйте Jitpack , строки для добавления почти идентичны по сравнению с Groovy. Вы можете вдохновить меня на этот проект .

Прочитайте Настройка сборки Kotlin онлайн: https://riptutorial.com/ru/kotlin/topic/2501/настройка-сборки-kotlin

глава 24: Нулевая безопасность

Examples

Nullable и Non-Nullable типы

Обычные типы, такие как `String` , не имеют значения NULL. Чтобы сделать их способными удерживать нулевые значения, вы должны явно обозначить, что, поставив а `?` за ними: `String?`

```
var string        : String = "Hello World!"
var nullableString: String? = null

string = nullableString    // Compiler error: Can't assign nullable to non-nullable type.
nullableString = string    // This will work however!
```

Оператор безопасного вызова

Для доступа к функциям и свойствам типов с нулевым значением вам необходимо использовать специальные операторы.

Первый, `?.` , дает вам свойство или функцию, к которой вы пытаетесь получить доступ, или дает null, если объект имеет значение null:

```
val string: String? = "Hello World!"
print(string.length)   // Compile error: Can't directly access property of nullable type.
print(string?.length)  // Will print the string's length, or "null" if the string is null.
```

Идиома: вызов нескольких методов на одном и том же объекте с нулевой отметкой

Элегантный способ назвать несколько методов нулевой проверкой объекта с использованием Котлин - х `apply` так:

```
obj?.apply {
    foo()
    bar()
}
```

Это вызовет `foo` и `bar` на `obj` (`this` в блоке `apply`), только если `obj` является нулевым, в противном случае пропускает весь блок.

Чтобы принести переменную с нулевым значением в область видимости как ссылку, не содержащую нулевых значений, не делая ее неявным приемом вызовов функций и свойств,

вы можете использовать `let` вместо `apply` :

```
nullable?.let { notnull ->
    notnull.foo()
    notnull.bar()
}
```

`notnull` можно было бы назвать что - нибудь, или даже оставил, и использовать через неявного параметра лямбда `it` .

Смарт-броски

Если компилятор может сделать вывод о том, что объект не может иметь значение null в определенный момент, вам больше не нужно использовать специальные операторы:

```
var string: String? = "Hello!"
print(string.length)  // Compile error
if(string != null) {
    // The compiler now knows that string can't be null
    print(string.length)  // It works now!
}
```

> **Примечание** . Компилятор не позволит вам использовать измененные переменные, которые могут быть изменены между нулевой проверкой и предполагаемым использованием.
>
> Если переменная доступна из-за пределов области текущего блока (например, они являются членами нелокального объекта), вам нужно создать новую локальную ссылку, которую затем вы сможете использовать и использовать.

Устранение нулей из Iterable и массива

Иногда нам нужно изменить тип из `Collection<T?> Collections<T>` . В этом случае `filterNotNull` является нашим решением.

```
val a: List<Int?> = listOf(1, 2, 3, null)
val b: List<Int> = a.filterNotNull()
```

Null Coalescing / Elvis Operator

Иногда желательно оценивать значение NULL в стиле if-else. Оператор Элвиса, `?:` , Может быть использован в Котлине для такой ситуации.

Например:

```
val value: String = data?.first() ?: "Nothing here."
```

Вышеприведенное выражение возвращает `"Nothing here"` если `data?.first()` или сами `data` дают `null` значение иначе результат `data?.first()` .

Кроме того, можно исключить исключения, используя тот же синтаксис, чтобы прервать выполнение кода.

```
val value: String = data?.second()
    ?: throw IllegalArgumentException("Value can't be null!")
```

> Напоминание: NullPointerExceptions могут быть выбрасываться с помощью оператора утверждения (например, `data!!.second()!!`)

Утверждение

`!!` суффиксы игнорируют значение nullability и возвращают ненулевую версию этого типа. `KotlinNullPointerException` будет `KotlinNullPointerException` , если объект является `null` .

```
val message: String? = null
println(message!!) //KotlinNullPointerException thrown, app crashes
```

Оператор Элвиса (? :)

В Kotlin мы можем объявить переменную, которая может содержать `null reference` . Предположим, что мы имеем нулевую ссылку `a` , мы можем сказать: «если `a` не является нулевым, используйте его, в противном случае используйте некоторое ненулевое значение `x` »

```
var a: String? = "Nullable String Value"
```

Теперь `a` может быть нулевым. Поэтому, когда нам нужно получить доступ к значению `a` , нам нужно выполнить проверку безопасности, независимо от того, содержит ли она значение или нет. Мы можем выполнить эту проверку безопасности с помощью обычного выражения `if...else` .

```
val b: Int = if (a != null) a.length else -1
```

Но здесь идет вперед оператор `Elvis` (Elvis Оператор: `?:`). Выше, `if...else` может быть выражено с помощью оператора Элвиса, как показано ниже:

```
val b = a?.length ?: -1
```

Если выражение слева от `?:` (Здесь: `a?.length`) не равно null, оператор elvis возвращает его, иначе он возвращает выражение вправо (здесь: `-1`). Выражение правой стороны оценивается только в том случае, если левая сторона равна нулю.

Прочитайте Нулевая безопасность онлайн: https://riptutorial.com/ru/kotlin/topic/2080/нулевая-безопасность

глава 25: Объекты Singleton

Вступление

Объектом является особый вид класса, который может быть объявлен с использованием ключевого слова `object` . Объекты похожи на Singletons (шаблон дизайна) в java. Он также функционирует как статическая часть java. Начинающие, которые переключаются с java на kotlin, могут широко использовать эту функцию вместо статических или синглетонов.

Examples

Использовать в качестве репликации статических методов / полей java

```
object CommonUtils {

    var anyname: String ="Hello"

    fun dispMsg(message: String) {
        println(message)
    }
}
```

Из любого другого класса просто вызовите переменную и функции следующим образом:

```
CommonUtils.anyname
CommonUtils.dispMsg("like static call")
```

Использовать в качестве одноэлементного

Объекты Котлина на самом деле представляют собой только одноточие. Его основным преимуществом является то, что вам не нужно использовать `SomeSingleton.INSTANCE` чтобы получить экземпляр singleton.

В java ваш синглтон выглядит следующим образом:

```
public enum SharedRegistry {
    INSTANCE;
    public void register(String key, Object thing) {}
}

public static void main(String[] args) {
    SharedRegistry.INSTANCE.register("a", "apple");
    SharedRegistry.INSTANCE.register("b", "boy");
    SharedRegistry.INSTANCE.register("c", "cat");
    SharedRegistry.INSTANCE.register("d", "dog");
}
```

В котлине эквивалентный код

```
object SharedRegistry {
    fun register(key: String, thing: Object) {}
}

fun main(Array<String> args) {
    SharedRegistry.register("a", "apple")
    SharedRegistry.register("b", "boy")
    SharedRegistry.register("c", "cat")
    SharedRegistry.register("d", "dog")
}
```

Это obvoiusly менее подробный для использования.

Прочитайте Объекты Singleton онлайн: https://riptutorial.com/ru/kotlin/topic/10152/объекты-singleton

глава 26: Основы Котлина

Вступление

В этом разделе рассказывается об основах Котлина для начинающих.

замечания

1. Файл Kotlin имеет расширение .kt.
2. Все классы в Kotlin имеют общий суперкласс All, который является суперполе по умолчанию для класса без объявленных супертипов (аналогично Object in Java).
3. Переменные могут быть объявлены как val (неизменяемый-присваивать один раз) или var (изменяемое значение может быть изменено)
4. Точка с запятой не нужна в конце инструкции.
5. Если функция не возвращает какое-либо полезное значение, его возвращаемым типом является Unit.It также является необязательным. 6. Условие равенства проверяется операцией ===. a === b оценивает значение true тогда и только тогда, когда a и b указывают на один и тот же объект.

Examples

Основные примеры

Декларация типа декларации единицы является необязательной для функций. Следующие коды эквивалентны.

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
}

fun printHello(name: String?) {
    ...
}
```

Функции 2.Single-Expression: когда функция возвращает одно выражение, фигурные скобки могут быть опущены, а тело указано после символа =

```
fun double(x: Int): Int = x * 2
```

Явное объявление типа возврата является необязательным, если это может быть выведено компилятором

```
fun double(x: Int) = x * 2
```

Интерполяция 3.String: использование строковых значений легко.

```
In java:
    int num=10
    String s  = "i =" + i;

In Kotlin
    val num = 10
    val s = "i = $num"
```

4. В Kotlin система типов различает ссылки, которые могут содержать нуль (нулевые ссылки) и те, которые не могут (непустые ссылки). Например, регулярная переменная типа String не может иметь значение null:

```
var a: String = "abc"
a = null // compilation error
```

Чтобы разрешить nulls, мы можем объявить переменную как строку с нулевым значением, записанную String ?:

```
var b: String? = "abc"
b = null // ok
```

5. В Kotlin, == фактически проверяет равенство значений. В соответствии с соглашением выражение типа a == b переводится на

```
a?.equals(b) ?: (b === null)
```

Прочитайте Основы Котлина онлайн: https://riptutorial.com/ru/kotlin/topic/10648/основы-котлина

глава 27: отражение

Вступление

Отражение - это способность языка проверять код во время выполнения, а не на время компиляции.

замечания

Отражение - это механизм для интроспекции языковых конструкций (классов и функций) во время выполнения.

При таргетинге на платформу JVM функции отображения времени выполнения распределяются в отдельном JAR: `kotlin-reflect.jar` . Это делается для уменьшения размера исполняемого файла, сокращения неиспользуемых функций и возможности для таргетинга на другие (например, JS) платформы.

Examples

Ссылка на класс

Чтобы получить ссылку на объект `KClass` представляющий некоторый класс, используйте двойные двоеточия:

```
val c1 = String::class
val c2 = MyClass::class
```

Ссылка на функцию

Функции - первоклассные граждане в Котлине. Вы можете получить ссылку на нее, используя двойные двоеточия, а затем передать ее другой функции:

```
fun isPositive(x: Int) = x > 0

val numbers = listOf(-2, -1, 0, 1, 2)
println(numbers.filter(::isPositive)) // [1, 2]
```

Взаимодействие с отражением Java

Чтобы получить объект `Class` Java от `KClass` Kotlin, используйте свойство расширения `.java` :

```
val stringKClass: KClass<String> = String::class
val c1: Class<String> = stringKClass.java
```

```
val c2: Class<MyClass> = MyClass::class.java
```

Последний пример будет оптимизирован компилятором, чтобы не выделять промежуточный экземпляр `KClass` .

Получение значений всех свойств класса

Данный `Example` класса, расширяющий класс `BaseExample` с некоторыми свойствами:

```
open class BaseExample(val baseField: String)

class Example(val field1: String, val field2: Int, baseField: String):
    BaseExample(baseField) {

    val field3: String
        get() = "Property without backing field"

    val field4 by lazy { "Delegated value" }

    private val privateField: String = "Private value"
}
```

Можно получить все свойства класса:

```
val example = Example(field1 = "abc", field2 = 1, baseField = "someText")

example::class.memberProperties.forEach { member ->
    println("${member.name} -> ${member.get(example)}")
}
```

Запуск этого кода вызовет исключение. Свойство `private val privateField` объявляется приватным, а вызов `member.get(example)` на нем не будет выполнен. Один из способов справиться с этим - отфильтровать частные свойства. Для этого нам нужно проверить модификатор видимости Java getter. В случае `private val` геттер не существует, поэтому мы можем принять частный доступ.

Вспомогательная функция и ее использование могут выглядеть так:

```
fun isFieldAccessible(property: KProperty1<*, *>): Boolean {
    return property.javaGetter?.modifiers?.let { !Modifier.isPrivate(it) } ?: false
}

val example = Example(field1 = "abc", field2 = 1, baseField = "someText")

example::class.memberProperties.filter { isFieldAccessible(it) }.forEach { member ->
    println("${member.name} -> ${member.get(example)}")
}
```

Другой подход заключается в том, чтобы сделать частные свойства доступными с использованием рефлексии:

```
example::class.memberProperties.forEach { member ->
    member.isAccessible = true
    println("${member.name} -> ${member.get(example)}")
}
```

Установка значений всех свойств класса

В качестве примера мы хотим установить все свойства строки класса образца

```
class TestClass {
    val readOnlyProperty: String
        get() = "Read only!"

    var readWriteString = "asd"
    var readWriteInt = 23

    var readWriteBackedStringProperty: String = ""
        get() = field + '5'
        set(value) { field = value + '5' }

    var readWriteBackedIntProperty: Int = 0
        get() = field + 1
        set(value) { field = value - 1 }

    var delegatedProperty: Int by TestDelegate()

    private var privateProperty = "This should be private"

    private class TestDelegate {
        private var backingField = 3

        operator fun getValue(thisRef: Any?, prop: KProperty<*>): Int {
            return backingField
        }

        operator fun setValue(thisRef: Any?, prop: KProperty<*>, value: Int) {
            backingField += value
        }
    }
}
```

Получение изменчивых свойств основывается на получении всех свойств, фильтрации изменяемых свойств по типу. Нам также необходимо проверить видимость, так как чтение частных свойств приводит к исключению времени выполнения.

```
val instance = TestClass()
TestClass::class.memberProperties
        .filter{ prop.visibility == KVisibility.PUBLIC }
        .filterIsInstance<KMutableProperty<*>>()
        .forEach { prop ->
            System.out.println("${prop.name} -> ${prop.get(instance)")
        }
```

Чтобы установить все свойства `String` в `"Our Value"` мы можем дополнительно фильтровать тип возвращаемого значения. Поскольку Kotlin основан на Java VM, тип Erasure действует,

и, таким образом, свойства, возвращающие общие типы, такие как `List<String>` будут такими же, как `List<Any>` . Печальное отражение - не золотая пуля, и нет разумного способа избежать этого, поэтому вам нужно следить в своих случаях использования.

```
val instance = TestClass()
TestClass::class.memberProperties
        .filter{ prop.visibility == KVisibility.PUBLIC }
        // We only want strings
        .filter{ it.returnType.isSubtypeOf(String::class.starProjectedType) }
        .filterIsInstance<KMutableProperty<*>>()
        .forEach { prop ->
            // Instead of printing the property we set it to some value
            prop.setter.call(instance, "Our Value")
        }
```

Прочитайте отражение онлайн: https://riptutorial.com/ru/kotlin/topic/2402/отражение

глава 28: Петли в Котлине

замечания

В Котлине петли собираются вплоть до оптимизированных петель, где это возможно. Например, если вы перебираете диапазон чисел, байт-код будет скомпилирован до соответствующего цикла на основе простых значений int, чтобы избежать накладных расходов на создание объекта.

Examples

Повторить действие х раз

```
repeat(10) { i ->
    println("This line will be printed 10 times")
    println("We are on the ${i + 1}. loop iteration")
}
```

Пересечение повторяющихся объектов

Вы можете циклически перебирать любые итерации с помощью стандартного цикла for:

```
val list = listOf("Hello", "World", "!")
for(str in list) {
    print(str)
}
```

В Котлине много вещей, таких как числовые диапазоны:

```
for(i in 0..9) {
    print(i)
}
```

Если вам нужен индекс во время итерации:

```
for((index, element) in iterable.withIndex()) {
    print("$element at index $index")
}
```

Существует также функциональный подход к итерации, включенный в стандартную библиотеку, без видимых языковых конструкций, с использованием функции forEach:

```
iterable.forEach {
    print(it.toString())
}
```

`it` в этом примере неявно удерживает текущий элемент, см. Лямбда-функции

В то время как петли

В то время как циклы do-while работают так же, как на других языках:

```
while(condition) {
    doSomething()
}

do {
    doSomething()
} while (condition)
```

В цикле do-while блок условий имеет доступ к значениям и переменным, объявленным в теле цикла.

Перерыв и продолжение

Перерыв и продолжение ключевых слов работают так же, как на других языках.

```
while(true) {
    if(condition1) {
        continue // Will immediately start the next iteration, without executing the rest of
the loop body
    }
    if(condition2) {
        break // Will exit the loop completely
    }
}
```

Если у вас есть вложенные циклы, вы можете пометить операторы цикла и квалифицировать инструкции break и continue, чтобы указать, какой цикл вы хотите продолжить или разбить:

```
outer@ for(i in 0..10) {
    inner@ for(j in 0..10) {
        break       // Will break the inner loop
        break@inner // Will break the inner loop
        break@outer // Will break the outer loop
    }
}
```

Однако этот подход не будет работать для функциональной для `forEach` конструкции.

Итерация по карте в Котлине

```
//iterates over a map, getting the key and value at once

var map = hashMapOf(1 to "foo", 2 to "bar", 3 to "baz")
```

```
for ((key, value) in map) {
    println("Map[$key] = $value")
}
```

Рекурсия

Циклирование через рекурсию также возможно в Котлине, как и в большинстве языков программирования.

```
fun factorial(n: Long): Long = if (n == 0) 1 else n * factorial(n - 1)

println(factorial(10)) // 3628800
```

В приведенном выше примере `factorial` функция будет вызываться повторно сама по себе до тех пор, пока не будет выполнено заданное условие.

Функциональные конструкции для итерации

Стандартная библиотека Kotlin также предоставляет множество полезных функций для итеративной работы над коллекциями.

Например, функция `map` может использоваться для преобразования списка элементов.

```
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
val numberStrings = numbers.map { "Number $it" }
```

Одним из многих преимуществ этого стиля является то, что он позволяет осуществлять операции в цепочке аналогичным образом. Может потребоваться только незначительная модификация, если, скажем, список, указанный выше, необходимо отфильтровать для четных чисел. Можно использовать функцию `filter` .

```
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
val numberStrings = numbers.filter { it % 2 == 0 }.map { "Number $it" }
```

Прочитайте Петли в Котлине онлайн: https://riptutorial.com/ru/kotlin/topic/2727/петли-в-котлине

глава 29: Предостережения Котлина

Examples

Вызов toString () для типа NULL

Что следует учитывать при использовании метода `toString` в Kotlin - это обработка null в сочетании со `String?` ,

Например, вы хотите получить текст из `EditText` в Android.

У вас будет код:

```
// Incorrect:
val text = view.textField?.text.toString() ?: ""
```

Вы ожидаете, что если поле не существует, значение будет пустой строкой, но в этом случае оно будет `"null"` .

```
// Correct:
val text = view.textField?.text?.toString() ?: ""
```

Прочитайте Предостережения Котлина онлайн: https://riptutorial.com/ru/kotlin/topic/6608/предостережения-котлина

глава 30: регистрация в котлин

замечания

Связанный вопрос: Идиоматический способ регистрации в Котлине

Examples

kotlin.logging

```
class FooWithLogging {
  companion object: KLogging()

  fun bar() {
    logger.info { "hello $name" }
  }

  fun logException(e: Exception) {
    logger.error(e) { "Error occured" }
  }
}
```

Использование kotlin.logging framework

Прочитайте регистрация в котлин онлайн: https://riptutorial.com/ru/kotlin/topic/3258/регистрация-в-котлин

глава 31: сопрограммы

Вступление

Примеры экспериментальной (пока) реализации Ковтонов

Examples

Простая сопрограмма, которая задерживает 1 секунду, но не блокирует

(из официального документа)

```
fun main(args: Array<String>) {
    launch(CommonPool) { // create new coroutine in common thread pool
        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
        println("World!") // print after delay
    }
    println("Hello,") // main function continues while coroutine is delayed
    Thread.sleep(2000L) // block main thread for 2 seconds to keep JVM alive
}
```

результат

```
Hello,
World!
```

Прочитайте сопрограммы онлайн: https://riptutorial.com/ru/kotlin/topic/10936/сопрограммы

глава 32: Строительство DSL

Вступление

Сосредоточьтесь на деталях синтаксиса для проектирования внутренних DSL в Котлине.

Examples

Подход Infix для создания DSL

Если у вас есть:

```
infix fun <T> T?.shouldBe(expected: T?) = assertEquals(expected, this)
```

вы можете написать следующий код DSL-типа в своих тестах:

```
@Test
fun test() {
  100.plusOne() shouldBe 101
}
```

Переопределение метода вызова для создания DSL

Если у вас есть:

```
class MyExample(val i: Int) {
  operator fun <R> invoke(block: MyExample.() -> R) = block()
  fun Int.bigger() = this > i
}
```

вы можете написать следующий код DSL-кода в своем производственном коде:

```
fun main2(args: Array<String>) {
    val ex = MyExample(233)
    ex {
        // bigger is defined in the context of `ex`
        // you can only call this method inside this context
        if (777.bigger()) kotlin.io.println("why")
    }
}
```

Использование операторов с лямбдами

Если у вас есть:

```
val r = Random(233)
```

```
infix inline operator fun Int.rem(block: () -> Unit) {
  if (r.nextInt(100) < this) block()
}
```

Вы можете написать следующий DSL-код:

```
20 % { println("The possibility you see this message is 20%") }
```

Использование расширений с lambdas

Если у вас есть:

```
operator fun <R> String.invoke(block: () -> R) = {
  try { block.invoke() }
  catch (e: AssertException) { System.err.println("$this\n${e.message}") }
}
```

Вы можете написать следующий DSL-код:

```
"it should return 2" {
   parse("1 + 1").buildAST().evaluate() shouldBe 2
}
```

Если вы чувствуете смущение с `shouldBe` выше, см. Пример `Infix approach to build DSL` .

Прочитайте Строительство DSL онлайн: https://riptutorial.com/ru/kotlin/topic/10042/строительство-dsl

глава 33: Струны

Examples

Элементы строки

Элементы String - это символы, к которым можно получить доступ с помощью `string[index]` операции индексирования `string[index]` .

```
val str = "Hello, World!"
println(str[1]) // Prints e
```

Элементы String можно повторить с помощью цикла for.

```
for (c in str) {
    println(c)
}
```

Строковые литералы

Котлин имеет два типа строковых литералов:

- Снятая строка
- Исходная строка

Сбежавшая строка обрабатывает специальные символы, экранируя их. Экранирование выполняется с обратной косой чертой. Поддерживаются следующие escape-последовательности: `\t` , `\b` , `\n` , `\r` , `\'` , `\"` , `\\` и `\$` . Для кодирования любого другого символа используйте синтаксис escape-последовательности Unicode: `\uFF00` .

```
val s = "Hello, world!\n"
```

Исходная строка, разделенная тройной цитатой `"""` , не содержит экранирования и может содержать символы новой строки и любые другие символы

```
val text = """
    for (c in "foo")
        print(c)
"""
```

Ведущие пробелы можно удалить с помощью функции trimMargin () .

```
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
```

```
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

Префикс префикса по умолчанию - символ канала `|` , это может быть задано как параметр trimMargin; например `trimMargin(">")` .

Строковые шаблоны

Обе экранированные строки и необработанные строки могут содержать шаблонные выражения. Выражение шаблона - это фрагмент кода, который оценивается, и его результат объединяется в строку. Он начинается со знака доллара `$` и состоит либо из имени переменной:

```
val i = 10
val s = "i = $i" // evaluates to "i = 10"
```

Или произвольное выражение в фигурных скобках:

```
val s = "abc"
val str = "$s.length is ${s.length}" // evaluates to "abc.length is 3"
```

Чтобы включить буквальный знак доллара в строку, убегите с ним, используя обратную косую черту:

```
val str = "\$foo" // evaluates to "$foo"
```

Исключение составляют необработанные строки, которые не поддерживают экранирование. В необработанных строках вы можете использовать следующий синтаксис для обозначения знака доллара.

```
val price = """
${'$'}9.99
"""
```

Строковое равенство

В котлинских строках сравниваются с `==` оператором, которые чётствуют за их структурное равенство.

```
val str1 = "Hello, World!"
val str2 = "Hello," + " World!"
println(str1 == str2) // Prints true
```

Ссылочное равенство проверяется с помощью оператора `===` .

```
val str1 = """
```

```
    |Hello, World!
    """.trimMargin()

val str2 = """
    #Hello, World!
    """.trimMargin("#")

val str3 = str1

println(str1 == str2) // Prints true
println(str1 === str2) // Prints false
println(str1 === str3) // Prints true
```

Прочитайте Струны онлайн: https://riptutorial.com/ru/kotlin/topic/8285/струны

глава 34: Тип псевдонимов

Вступление

С помощью псевдонимов типов мы можем присвоить псевдоним другому типу. Он идеален для присвоения имени таким типам функций, как `(String) -> Boolean` или generic type типа `Pair<Person, Person>` .

Типичные псевдонимы поддерживают дженерики. Псевдоним может заменить тип дженериками, а псевдоним может быть дженериками.

Синтаксис

- **typealias** *alias-name* **=** *существующий тип*

замечания

Тип псевдонимов является особенностью компилятора. В сгенерированном коде для JVM ничего не добавляется. Все псевдонимы будут заменены реальным типом.

Examples

Тип функции

```
typealias StringValidator = (String) -> Boolean
typealias Reductor<T, U, V> = (T, U) -> V
```

Общий тип

```
typealias Parents = Pair<Person, Person>
typealias Accounts = List<Account>
```

Прочитайте Тип псевдонимов онлайн: https://riptutorial.com/ru/kotlin/topic/9453/тип-псевдонимов

глава 35: Тип-безопасные строители

замечания

Тип-безопасный строитель представляет собой концепцию, а не функцию языка, поэтому она не является строго формализованной.

Типичная структура типа безопасного строителя

Одна функция-строитель обычно состоит из трех шагов:

1. Создайте объект.
2. Выполните lambda для инициализации объекта.
3. Добавьте объект в структуру или верните его.

Тип-безопасные сборщики в библиотеках Котлин

Концепция типов безопасных строителей широко используется в некоторых библиотеках и структурах Котлин, например:

- Анко
- Wasabi
- Ktor
- спекуляция

Examples

Тип-древовидный строитель

Строителей можно определить как набор функций расширения, принимающих лямбда-выражения с приемниками в качестве аргументов. В этом примере создается меню `JFrame` :

```
import javax.swing.*

fun JFrame.menuBar(init: JMenuBar.() -> Unit) {
    val menuBar = JMenuBar()
    menuBar.init()
    setJMenuBar(menuBar)
}

fun JMenuBar.menu(caption: String, init: JMenu.() -> Unit) {
    val menu = JMenu(caption)
    menu.init()
    add(menu)
}
```

```
fun JMenu.menuItem(caption: String, init: JMenuItem.() -> Unit) {
    val menuItem = JMenuItem(caption)
    menuItem.init()
    add(menuItem)
}
```

Эти функции затем могут быть использованы для создания древовидной структуры объектов простым способом:

```
class MyFrame : JFrame() {
    init {
        menuBar {
            menu("Menu1") {
                menuItem("Item1") {
                    // Initialize MenuItem with some Action
                }
                menuItem("Item2") {}
            }
            menu("Menu2") {
                menuItem("Item3") {}
                menuItem("Item4") {}
            }
        }
    }
}
```

Прочитайте Тип-безопасные строители онлайн: https://riptutorial.com/ru/kotlin/topic/6010/тип-безопасные-строители

глава 36: Условные заявления

замечания

В отличие от `switch` Java, оператор `when` не имеет никакого сквозного поведения. Это означает, что если ветвь согласована, поток управления возвращается после его выполнения, и оператор `break` не требуется. Если вы хотите объединить bahaviors для нескольких аргументов, вы можете написать несколько аргументов, разделенных запятыми:

```
when (x) {
    "foo", "bar" -> println("either foo or bar")
    else -> println("didn't match anything")
}
```

Examples

Стандартный if-statement

```
val str = "Hello!"
if (str.length == 0) {
    print("The string is empty!")
} else if (str.length > 5) {
    print("The string is short!")
} else {
    print("The string is long!")
}
```

В противном случае else-ветви являются необязательными в нормальных операторах if.

Оператор if в качестве выражения

Операторы if могут быть выражениями:

```
val str = if (condition) "Condition met!" else "Condition not met!"
```

Обратите внимание, что `else` -branch не является необязательным, если `if` -statement используется как выражение.

Это также можно сделать с помощью многострочного варианта с фигурными скобками и несколькими `else if` .

```
val str = if (condition1){
    "Condition1 met!"
   } else if (condition2) {
    "Condition2 met!"
```

```
    } else {
     "Conditions not met!"
    }
```

> СОВЕТ: Kotlin может вывести тип переменной для вас, но если вы хотите быть уверенным в типе, просто комментируйте его по переменной, как: `val str: String =` это обеспечит соблюдение типа и упростит его чтение.

Оператор when вместо цепочек if-else-if

Оператор while является альтернативой if-statement с несколькими else-if-ветвями:

```
when {
    str.length == 0 -> print("The string is empty!")
    str.length > 5  -> print("The string is short!")
    else            -> print("The string is long!")
}
```

Тот же код, написанный с использованием цепочки *if-else-if* :

```
if (str.length == 0) {
    print("The string is empty!")
} else if (str.length > 5) {
    print("The string is short!")
} else {
    print("The string is long!")
}
```

Как и в случае if-statement, else-branch является необязательным, и вы можете добавить столько или несколько ветвей, сколько хотите. Вы также можете иметь многострочные ветви:

```
when {
    condition -> {
        doSomething()
        doSomeMore()
    }
    else -> doSomethingElse()
}
```

Согласование аргументов-оператора

При задании аргумента, `when` -statement соответствует аргументу против ветвей в последовательности. Согласование выполняется с помощью оператора `==` который выполняет нулевые проверки и сравнивает операнды с помощью функции `equals` . Первый соответствующий будет выполнен.

```
when (x) {
    "English" -> print("How are you?")
    "German" -> print("Wie geht es dir?")
```

```
    else -> print("I don't know that language yet :(")
}
```

Оператор when также знает некоторые дополнительные варианты соответствия:

```
val names = listOf("John", "Sarah", "Tim", "Maggie")
when (x) {
    in names -> print("I know that name!")
    !in 1..10 -> print("Argument was not in the range from 1 to 10")
    is String -> print(x.length) // Due to smart casting, you can use String-functions here
}
```

Когда выражение-выражение как выражение

Как если бы, когда также можно было использовать в качестве выражения:

```
val greeting = when (x) {
    "English" -> "How are you?"
    "German" -> "Wie geht es dir?"
    else -> "I don't know that language yet :("
}
print(greeting)
```

Чтобы использоваться как выражение, оператор when должен быть исчерпывающим, то есть иметь либо ветку else или покрывать все возможности ветвями по-другому.

Когда-заявление с перечислениями

`when` их можно использовать для соответствия значениям `enum` :

```
enum class Day {
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}

fun doOnDay(day: Day) {
    when(day) {
        Day.Sunday ->     // Do something
        Day.Monday, Day.Tuesday ->     // Do other thing
        Day.Wednesday ->  // ...
        Day.Thursday ->   // ...
        Day.Friday ->     // ...
        Day.Saturday ->   // ...
    }
}
```

Как вы можете видеть во второй строке (`Monday` и `Tuedsay`), также возможно объединить два или более значений `enum` .

Если ваши случаи не являются исчерпывающими, компиляция покажет ошибку. Вы можете использовать `else` для обработки случаев по умолчанию:

```
fun doOnDay(day: Day) {
    when(day) {
        Day.Monday ->      // Work
        Day.Tuesday ->     // Work hard
        Day.Wednesday ->   // ...
        Day.Thursday ->    //
        Day.Friday ->      //
        else ->            // Party on weekend
    }
}
```

Хотя то же самое можно сделать, используя конструкцию `if-then-else` , `when` берет на себя недостающие значения `enum` и делает ее более естественной.

Проверьте здесь дополнительную информацию о kotlin `enum`

Прочитайте Условные заявления онлайн: https://riptutorial.com/ru/kotlin/topic/2685/условные-заявления

глава 37: функции

Синтаксис

- fun ***Имя*** (***Params***) = ...
- fun ***Имя*** (***Params***) {...}
- fun ***Имя*** (***Params***): ***Введите*** {...}
- fun < ***Type Argument*** > ***Name*** (***Params***): ***Type*** {...}
- inline fun ***Имя*** (***Params***): ***Введите*** {...}
- { ***ArgName*** : ***ArgType*** -> ...}
- { ***ArgName*** -> ...}
- { ***ArgNames*** -> ...}
- {(***ArgName*** : ***ArgType***): ***Тип*** -> ...}

параметры

параметр	подробности
название	Название функции
Params	Значения, присвоенные функции с именем и типом: `Name : Type`
Тип	Тип возвращаемой функции
Тип Аргумент	Параметр типа, используемый в общем программировании (необязательно возвращаемый тип)
ArgName	Название значения, заданного функции
тип аргумента	***Спецификатор*** типа для ***ArgName***
ArgNames	Список ArgName, разделенный запятыми

Examples

Функции, выполняющие другие функции

Как видно из «Лямбда-функций», функции могут принимать другие функции в качестве параметра. «Тип функции», который вам потребуется для объявления функций, выполняющих другие функции, выглядит следующим образом:

```
# Takes no parameters and returns anything
() -> Any?

# Takes a string and an integer and returns ReturnType
(arg1: String, arg2: Int) -> ReturnType
```

Например, вы можете использовать нечеткий тип, `() -> Any?` , чтобы объявить функцию, которая выполняет лямбда-функцию дважды:

```
fun twice(x: () -> Any?) {
    x(); x();
}

fun main() {
    twice {
        println("Foo")
    } # => Foo
      # => Foo
}
```

Лямбда-функции

Лямбда-функции - это анонимные функции, которые обычно создаются во время вызова функции, чтобы действовать как параметр функции. Они объявляются окружающими выражениями с помощью {braces} - если нужны аргументы, они помещаются перед стрелкой `->` .

```
{ name: String ->
    "Your name is $name" //This is returned
}
```

Последним утверждением внутри лямбда-функции является автоматически возвращаемое значение.

Тип необязателен, если вы помещаете лямбду в место, где компилятор может вывести типы.

Несколько аргументов:

```
{ argumentOne:String, argumentTwo:String ->
    "$argumentOne - $argumentTwo"
}
```

Если функция лямбда нужен только один аргумент, то список аргументов может быть опущен , и единственный аргумент упоминаться , используя `it` вместо этого.

```
{ "Your name is $it" }
```

Если единственным аргументом функции является лямбда-функция, то круглые скобки

могут быть полностью исключены из вызова функции.

```
# These are identical
listOf(1, 2, 3, 4).map { it + 2 }
listOf(1, 2, 3, 4).map({ it + 2 })
```

Ссылки на функции

Мы можем ссылаться на функцию, не называя ее, префиксное имя функции с помощью `::` . Затем это можно передать функции, которая принимает какую-либо другую функцию в качестве параметра.

```
fun addTwo(x: Int) = x + 2
listOf(1, 2, 3, 4).map(::addTwo) # => [3, 4, 5, 6]
```

Функции без приемника будут преобразованы в `(ParamTypeA, ParamTypeB, ...) -> ReturnType` где `ParamTypeA` , `ParamTypeB` ... являются типом параметров функции, а `ReturnType1 - тип возвращаемого значения функции.

```
fun foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
println(::foo::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function3<Foo0, Foo1, Foo2, Bar>
// Human readable type: (Foo0, Foo1, Foo2) -> Bar
```

Функции с приемником (будь то функция расширения или функция-член) имеют другой синтаксис. Вы должны добавить имя типа приемника перед двойным двоеточием:

```
class Foo
fun Foo.foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
val ref = Foo::foo
println(ref::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function4<Foo, Foo0, Foo1, Foo2, Bar>
// Human readable type: (Foo, Foo0, Foo1, Foo2) -> Bar
// takes 4 parameters, with receiver as first and actual parameters following, in their order

// this function can't be called like an extension function, though
val ref = Foo::foo
Foo().ref(Foo0(), Foo1(), Foo2()) // compile error

class Bar {
    fun bar()
}
print(Bar::bar) // works on member functions, too.
```

Однако, когда приемник функции является объектом, приемник опускается из списка параметров, потому что это и есть только один экземпляр такого типа.

```
object Foo
fun Foo.foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
val ref = Foo::foo
println(ref::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function3<Foo0, Foo1, Foo2, Bar>
// Human readable type: (Foo0, Foo1, Foo2) -> Bar
// takes 3 parameters, receiver not needed

object Bar {
    fun bar()
}
print(Bar::bar) // works on member functions, too.
```

Так как kotlin 1.1, ссылка на функцию также может быть *ограничена* переменной, которая затем называется *ссылкой* на *ограниченную функцию* .

1.1.0

```
fun makeList(last: String?): List<String> {
    val list = mutableListOf("a", "b", "c")
    last?.let(list::add)
    return list
}
```

Обратите внимание, что этот пример приведен только для того, чтобы показать, как работает ограниченная ссылка на функцию. Это плохая практика во всех других смыслах.

Однако есть особый случай. Нельзя ссылаться на функцию расширения, объявленную как член.

```
class Foo
class Bar {
    fun Foo.foo() {}
    val ref = Foo::foo // compile error
}
```

Основные функции

Функции объявляются с использованием ключевого слова `fun` , за которым следует имя функции и любые параметры. Вы также можете указать тип возврата функции, которая по умолчанию относится к `Unit` . Тело функции заключено в фигурные скобки `{}` . Если тип возврата отличается от `Unit` , тело должно выдать оператор возврата для каждой завершающей ветви внутри тела.

```
fun sayMyName(name: String): String {
    return "Your name is $name"
}
```

Сокращенная версия того же:

```
fun sayMyName(name: String): String = "Your name is $name"
```

И тип может быть опущен, так как можно сделать вывод:

```
fun sayMyName(name: String) = "Your name is $name"
```

Сокращенные функции

Если функция содержит только одно выражение, мы можем опустить скобки скобок и вместо этого использовать равные значения, как назначение переменной. Результат выражения возвращается автоматически.

```
fun sayMyName(name: String): String = "Your name is $name"
```

Встроенные функции

Функции могут быть объявлены inline с помощью `inline` префикса, и в этом случае они действуют как макросы в C - вместо того, чтобы быть вызванными, они заменяются кодом тела функции во время компиляции. Это может привести к повышению производительности при некоторых обстоятельствах, главным образом там, где lambdas используются в качестве функциональных параметров.

```
inline fun sayMyName(name: String) = "Your name is $name"
```

Одно отличие от макросов C состоит в том, что встроенные функции не могут получить доступ к области, из которой они вызваны:

```
inline fun sayMyName() = "Your name is $name"

fun main() {
    val name = "Foo"
    sayMyName() # => Unresolved reference: name
}
```

Функции оператора

Kotlin позволяет нам предоставлять реализации для предопределенного набора операторов с фиксированным символическим представлением (например, `+` или `*`) и фиксированным приоритетом. Для реализации оператора мы предоставляем функцию-член или функцию расширения с фиксированным именем для соответствующего типа. Функции, которые перегружают операторы, должны быть отмечены модификатором `operator` :

```
data class IntListWrapper (val wrapped: List<Int>) {
    operator fun get(position: Int): Int = wrapped[position]
}
```

```
val a = IntListWrapper(listOf(1, 2, 3))
a[1] // == 2
```

Дополнительные функции оператора можно найти в здесь

Прочитайте функции онлайн: https://riptutorial.com/ru/kotlin/topic/1280/функции

глава 38: Эквиваленты потока Java 8

Вступление

Kotlin предоставляет множество методов расширения для коллекций и итераций для применения функциональных операций. Специальный тип `Sequence` позволяет использовать ленивый состав нескольких таких операций.

замечания

О лени

Если вы хотите лениво обрабатывать цепочку, вы можете преобразовать ее в `Sequence` используя `asSequence()` перед цепочкой. В конце цепочки функций вы обычно получаете также `Sequence` . Затем вы можете использовать `toList()` , `toSet()` , `toMap()` или какую-либо другую функцию для материализации `Sequence` в конце.

```
// switch to and from lazy
val someList = items.asSequence().filter { ... }.take(10).map { ... }.toList()

// switch to lazy, but sorted() brings us out again at the end
val someList = items.asSequence().filter { ... }.take(10).map { ... }.sorted()
```

Почему нет типов?!?

Вы заметите, что примеры Kotlin не указывают типы. Это связано с тем, что Kotlin имеет полный вывод типа и полностью безопасен во время компиляции. Более того, чем Java, потому что он также имеет типы с нулевым значением и может помочь предотвратить опасный NPE. Так это в Котлине:

```
val someList = people.filter { it.age <= 30 }.map { it.name }
```

такой же как:

```
val someList: List<String> = people.filter { it.age <= 30 }.map { it.name }
```

Поскольку Котлин знает, что такое `people` , и что `people.age` является `Int` поэтому выражение фильтра допускает сравнение с `Int` и что `people.name` является `String` поэтому на этапе `map` создается `List<String>` (только для чтения `List String`).

Теперь, если `people` могут быть `null` , как в `List<People>?` затем:

```
val someList = people?.filter { it.age <= 30 }?.map { it.name }
```

Возвращает `List<String>?` который должен быть нулевым (*или использовать один из других операторов Котлина для значений с нулевым значением, см. этот идиоматический способ Kotlin для обработки значений с нулевым значением, а также идиоматический способ обработки нулевого или пустого списка в Котлине*)

Повторное использование потоков

В Котлине зависит от типа сбора, может ли он потребляться более одного раза. `Sequence` генерирует новый итератор каждый раз, и если он не утверждает, что «использует только один раз», он может каждый раз перезапускаться до начала каждого действия. Поэтому, если в потоке Java 8 не работает, но работает в Котлине:

```
// Java:
Stream<String> stream =
Stream.of("d2", "a2", "b1", "b3", "c").filter(s -> s.startsWith("b"));

stream.anyMatch(s -> true);    // ok
stream.noneMatch(s -> true);   // exception
```

```
// Kotlin:
val stream = listOf("d2", "a2", "b1", "b3", "c").asSequence().filter { it.startsWith('b' ) }

stream.forEach(::println) // b1, b2

println("Any B ${stream.any { it.startsWith('b') }}") // Any B true
println("Any C ${stream.any { it.startsWith('c') }}") // Any C false

stream.forEach(::println) // b1, b2
```

И в Java, чтобы получить такое же поведение:

```
// Java:
Supplier<Stream<String>> streamSupplier =
    () -> Stream.of("d2", "a2", "b1", "b3", "c")
          .filter(s -> s.startsWith("a"));

streamSupplier.get().anyMatch(s -> true);   // ok
streamSupplier.get().noneMatch(s -> true);  // ok
```

Поэтому в Котлине поставщик данных решает, может ли он вернуться и предоставить новый итератор или нет. Но если вы хотите преднамеренно ограничить `Sequence` для одноразовой итерации, вы можете использовать функцию `constrainOnce()` для `Sequence` следующим образом:

```
val stream = listOf("d2", "a2", "b1", "b3", "c").asSequence().filter { it.startsWith('b' ) }
        .constrainOnce()

stream.forEach(::println) // b1, b2
```

```
stream.forEach(::println) // Error:java.lang.IllegalStateException: This sequence can be
consumed only once.
```

Смотрите также:

- Справочник API для функций расширения для Iterable
- Ссылка API для функций расширения для массива
- Ссылка API для функций расширения для списка
- Ссылка API для функций расширения на карту

Examples

Накопить имена в списке

```
// Java:
List<String> list = people.stream().map(Person::getName).collect(Collectors.toList());
```

```
// Kotlin:
val list = people.map { it.name }  // toList() not needed
```

Преобразуйте элементы в строки и объедините их, разделяя их запятыми

```
// Java:
String joined = things.stream()
                       .map(Object::toString)
                       .collect(Collectors.joining(", "));
```

```
// Kotlin:
val joined = things.joinToString() // ", " is used as separator, by default
```

Вычислить сумму заработной платы работника

```
// Java:
int total = employees.stream()
                      .collect(Collectors.summingInt(Employee::getSalary)));
```

```
// Kotlin:
val total = employees.sumBy { it.salary }
```

Сотрудники группы по подразделениям

```
// Java:
Map<Department, List<Employee>> byDept
     = employees.stream()
                .collect(Collectors.groupingBy(Employee::getDepartment));
```

```
// Kotlin:
val byDept = employees.groupBy { it.department }
```

Вычисление суммы заработной платы отделом

```
// Java:
Map<Department, Integer> totalByDept
     = employees.stream()
                .collect(Collectors.groupingBy(Employee::getDepartment,
                     Collectors.summingInt(Employee::getSalary)));
```

```
// Kotlin:
val totalByDept = employees.groupBy { it.dept }.mapValues { it.value.sumBy { it.salary }}
```

Разделение студентов на прохождение и провал

```
// Java:
Map<Boolean, List<Student>> passingFailing =
     students.stream()
             .collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD));
```

```
// Kotlin:
val passingFailing = students.partition { it.grade >= PASS_THRESHOLD }
```

Имена мужчин-членов

```
// Java:
List<String> namesOfMaleMembersCollect = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .map(p -> p.getName())
    .collect(Collectors.toList());
```

```
// Kotlin:
val namesOfMaleMembers = roster.filter { it.gender == Person.Sex.MALE }.map { it.name }
```

Имена групп членов в реестре по полу

```
// Java:
Map<Person.Sex, List<String>> namesByGender =
      roster.stream().collect(
        Collectors.groupingBy(
            Person::getGender,
            Collectors.mapping(
                Person::getName,
                Collectors.toList())));
```

```
// Kotlin:
val namesByGender = roster.groupBy { it.gender }.mapValues { it.value.map { it.name } }
```

Отфильтровать список в другом списке

```
// Java:
List<String> filtered = items.stream()
    .filter( item -> item.startsWith("o") )
    .collect(Collectors.toList());
```

```
// Kotlin:
val filtered = items.filter { item.startsWith('o') }
```

Поиск кратчайшей строки списка

```
// Java:
String shortest = items.stream()
    .min(Comparator.comparing(item -> item.length()))
    .get();
```

```
// Kotlin:
val shortest = items.minBy { it.length }
```

Различные типы потоков №2 - лениво используя первый элемент, если существует

```
// Java:
Stream.of("a1", "a2", "a3")
    .findFirst()
    .ifPresent(System.out::println);
```

```
// Kotlin:
sequenceOf("a1", "a2", "a3").firstOrNull()?.apply(::println)
```

Различные типы потоков №3 - повторять ряд целых чисел

```
// Java:
IntStream.range(1, 4).forEach(System.out::println);
```

```
// Kotlin:  (inclusive range)
(1..3).forEach(::println)
```

Различные типы потоков # 4 - итерация массива, отображение значений, вычисление среднего значения

```
// Java:
Arrays.stream(new int[] {1, 2, 3})
    .map(n -> 2 * n + 1)
    .average()
    .ifPresent(System.out::println); // 5.0
```

```
// Kotlin:
arrayOf(1,2,3).map { 2 * it + 1}.average().apply(::println)
```

Различные типы потоков №5 - лениво перебирать список строк, отображать значения, преобразовывать в Int, находить max

```
// Java:
Stream.of("a1", "a2", "a3")
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .max()
    .ifPresent(System.out::println);  // 3
```

```
// Kotlin:
sequenceOf("a1", "a2", "a3")
    .map { it.substring(1) }
    .map(String::toInt)
    .max().apply(::println)
```

Различные типы потоков №6 - лениво перебирать поток интс, отображать значения, печатать результаты

```
// Java:
IntStream.range(1, 4)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

```
// Kotlin:  (inclusive range)
(1..3).map { "a$it" }.forEach(::println)
```

Различные типы потоков # 7 - лениво повторять парные числа, сопоставлять с Int, отображать на String, печатать каждый

```
// Java:
Stream.of(1.0, 2.0, 3.0)
    .mapToInt(Double::intValue)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

```
// Kotlin:
sequenceOf(1.0, 2.0, 3.0).map(Double::toInt).map { "a$it" }.forEach(::println)
```

Подсчет элементов в списке после применения фильтра

```
// Java:
long count = items.stream().filter( item -> item.startsWith("t")).count();
```

```
// Kotlin:
val count = items.filter { it.startsWith('t') }.size
// but better to not filter, but count with a predicate
val count = items.count { it.startsWith('t') }
```

Как работают потоки - фильтр, верхний регистр, затем сортировка списка

```
// Java:
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList.stream()
      .filter(s -> s.startsWith("c"))
      .map(String::toUpperCase)
     .sorted()
     .forEach(System.out::println);

// C1
// C2
```

```
// Kotlin:
val list = listOf("a1", "a2", "b1", "c2", "c1")
list.filter { it.startsWith('c') }.map (String::toUpperCase).sorted()
        .forEach (::println)
```

Различные типы потоков №1 - стремятся использовать первый элемент, если он существует

```
// Java:
Arrays.asList("a1", "a2", "a3")
    .stream()
    .findFirst()
    .ifPresent(System.out::println);
```

```
// Kotlin:
listOf("a1", "a2", "a3").firstOrNull()?.apply(::println)
```

или, создайте функцию расширения для строки, называемой ifPresent:

```
// Kotlin:
inline fun String?.ifPresent(thenDo: (String)->Unit) = this?.apply { thenDo(this) }

// now use the new extension function:
listOf("a1", "a2", "a3").firstOrNull().ifPresent(::println)
```

См. Также: Функция `apply()`

См. Также: Функции расширения

Смотрите также: `?.` Оператор безопасного вызова и, в общем, недействительность: http://stackoverflow.com/questions/34498562/in-kotlin-what-is-the-idiomatic-way-to-deal-with-nullable-values-referencing-o/34498563 # 34498563

Соберите пример № 5 - найдите людей, достигших совершеннолетия, выведите форматированную строку

```
// Java:
String phrase = persons
        .stream()
        .filter(p -> p.age >= 18)
        .map(p -> p.name)
        .collect(Collectors.joining(" and ", "In Germany ", " are of legal age."));

System.out.println(phrase);
// In Germany Max and Peter and Pamela are of legal age.
```

```
// Kotlin:
val phrase = persons
        .filter { it.age >= 18 }
        .map { it.name }
        .joinToString(" and ", "In Germany ", " are of legal age.")

println(phrase)
// In Germany Max and Peter and Pamela are of legal age.
```

И в качестве примечания в Kotlin мы можем создавать простые классы данных и создавать тестовые данные следующим образом:

```
// Kotlin:
// data class has equals, hashcode, toString, and copy methods automagically
data class Person(val name: String, val age: Int)

val persons = listOf(Person("Tod", 5), Person("Max", 33),
                     Person("Frank", 13), Person("Peter", 80),
                     Person("Pamela", 18))
```

Соберите пример # 6 - люди группы по возрасту, возраст печати и имена вместе

```
// Java:
Map<Integer, String> map = persons
        .stream()
        .collect(Collectors.toMap(
                p -> p.age,
                p -> p.name,
                (name1, name2) -> name1 + ";" + name2));
```

```
System.out.println(map);
// {18=Max, 23=Peter;Pamela, 12=David}
```

Хорошо, здесь больше интересного для Котлина. Сначала неправильные ответы, чтобы изучить варианты создания `Map` из коллекции / последовательности:

```
// Kotlin:
val map1 = persons.map { it.age to it.name }.toMap()
println(map1)
// output: {18=Max, 23=Pamela, 12=David}
// Result: duplicates overridden, no exception similar to Java 8

val map2 = persons.toMap({ it.age }, { it.name })
println(map2)
// output: {18=Max, 23=Pamela, 12=David}
// Result: same as above, more verbose, duplicates overridden

val map3 = persons.toMapBy { it.age }
println(map3)
// output: {18=Person(name=Max, age=18), 23=Person(name=Pamela, age=23), 12=Person(name=David,
age=12)}
// Result: duplicates overridden again

val map4 = persons.groupBy { it.age }
println(map4)
// output: {18=[Person(name=Max, age=18)], 23=[Person(name=Peter, age=23), Person(name=Pamela,
age=23)], 12=[Person(name=David, age=12)]}
// Result: closer, but now have a Map<Int, List<Person>> instead of Map<Int, String>

val map5 = persons.groupBy { it.age }.mapValues { it.value.map { it.name } }
println(map5)
// output: {18=[Max], 23=[Peter, Pamela], 12=[David]}
// Result: closer, but now have a Map<Int, List<String>> instead of Map<Int, String>
```

И теперь для правильного ответа:

```
// Kotlin:
val map6 = persons.groupBy { it.age }.mapValues { it.value.joinToString(";") { it.name } }

println(map6)
// output: {18=Max, 23=Peter;Pamela, 12=David}
// Result: YAY!!
```

Нам просто нужно было объединить соответствующие значения, чтобы свернуть списки и предоставить трансформатору `joinToString` для перехода от экземпляра `Person` к `Person.name` .

Соберите пример # 7a - Названия карт, объединяйтесь вместе с разделителем

```
// Java (verbose):
Collector<Person, StringJoiner, String> personNameCollector =
Collector.of(
        () -> new StringJoiner(" | "),          // supplier
```

```
        (j, p) -> j.add(p.name.toUpperCase()),  // accumulator
        (j1, j2) -> j1.merge(j2),               // combiner
        StringJoiner::toString);                // finisher

String names = persons
        .stream()
        .collect(personNameCollector);

System.out.println(names);  // MAX | PETER | PAMELA | DAVID

// Java (concise)
String names = persons.stream().map(p -> p.name.toUpperCase()).collect(Collectors.joining(" |
"));
```

```
// Kotlin:
val names = persons.map { it.name.toUpperCase() }.joinToString(" | ")
```

Соберите пример # 7b - Соберите с SummarizingInt

```
// Java:
IntSummaryStatistics ageSummary =
    persons.stream()
           .collect(Collectors.summarizingInt(p -> p.age));

System.out.println(ageSummary);
// IntSummaryStatistics{count=4, sum=76, min=12, average=19.000000, max=23}
```

```
// Kotlin:

// something to hold the stats...
data class SummaryStatisticsInt(var count: Int = 0,
                                var sum: Int = 0,
                                var min: Int = Int.MAX_VALUE,
                                var max: Int = Int.MIN_VALUE,
                                var avg: Double = 0.0) {
    fun accumulate(newInt: Int): SummaryStatisticsInt {
        count++
        sum += newInt
        min = min.coerceAtMost(newInt)
        max = max.coerceAtLeast(newInt)
        avg = sum.toDouble() / count
        return this
    }
}

// Now manually doing a fold, since Stream.collect is really just a fold
val stats = persons.fold(SummaryStatisticsInt()) { stats, person ->
stats.accumulate(person.age) }

println(stats)
// output: SummaryStatisticsInt(count=4, sum=76, min=12, max=23, avg=19.0)
```

Но лучше создать функцию расширения, 2 фактически соответствовать стилям в Kotlin stdlib:

```
// Kotlin:
```

```
inline fun Collection<Int>.summarizingInt(): SummaryStatisticsInt
        = this.fold(SummaryStatisticsInt()) { stats, num -> stats.accumulate(num) }

inline fun <T: Any> Collection<T>.summarizingInt(transform: (T)->Int): SummaryStatisticsInt =
        this.fold(SummaryStatisticsInt()) { stats, item -> stats.accumulate(transform(item)) }
```

Теперь у вас есть два способа использования новых функций `summarizingInt` :

```
val stats2 = persons.map { it.age }.summarizingInt()

// or

val stats3 = persons.summarizingInt { it.age }
```

И все они дают одинаковые результаты. Мы также можем создать это расширение для работы над `Sequence` и для соответствующих примитивных типов.

Прочитайте Эквиваленты потока Java 8 онлайн: https://riptutorial.com/ru/kotlin/topic/707/эквиваленты-потока-java-8

кредиты

S. No	Главы	Contributors
1	Начало работы с Kotlin	babedev, Community, cyberscientist, ganesshkumar, Ihor Kucherenko, Jayson Minard, mnoronha, neworld, Parker Hoyes, Ruckus T-Boom, Sach, Sean Reilly, Sheigutn, Simón Oroño, UnKnown, Urko Pineda
2	Enum	David Soroko, Kirill Rakhman, SerCe
3	JUnit	jenglert
4	Kotlin Android Extensions	Jemo Mgebrishvili, Ritave
5	Kotlin для разработчиков Java	Thorsten Schleinzer
6	RecyclerView в Котлине	Mohit Suthar
7	Regex	Espen, Travis
8	Аннотации	Brad Larson, Caelum, Héctor, Mood, piotrek1543, Sapan Zaveri
9	Базовый Лямбдас	memoizr, Rich Kuzsma
10	Варгарные параметры в функциях	byxor, piotrek1543, Sam
11	Делегирование класса	Sam
12	Делегированные свойства	Sam, Seaskyways
13	Дженерики	hotkey, Jayson Minard, KeksArmee
14	Идиомы	Aaron Christiansen, Adam Arold, Brad Larson, Héctor, Jayson Minard, Konrad Jamrozik, madhead, mayojava, razzledazzle, Sapan Zaveri, Serge Nikitin, yole
15	Изменяется	Nihal Saxena

16	Интерфейсы	Divya, Jan Vladimir Mostert, Jayson Minard, Ritave, Robin
17	Исключения	Brad Larson, jereksel, Sapan Zaveri
18	Коллекции	Ascension
19	Массивы	egor.zhdan, Sam, UnKnown
20	Методы расширения	Dávid Tímár, Jayson Minard, Kevin Robatel, Konrad Jamrozik, olivierlemasle, Parker Hoyes, razzledazzle
21	Модификаторы видимости	Avijit Karmakar
22	Наследование класса	byxor, KeksArmee, piotrek1543, Slav
23	Настройка сборки Kotlin	Aaron Christiansen, elect, madhead
24	Нулевая безопасность	KeksArmee, Kirill Rakhman, piotrek1543, razzledazzle, Robin, SerCe, Spidfire, technerd, Thorsten Schleinzer
25	Объекты Singleton	Divya, glee8e
26	Основы Котлина	Shinoo Goyal
27	отражение	atok, Kirill Rakhman, madhead, Ritave, Sup
28	Петли в Котлине	Ben Leggiero, JaseAnderson, mayojava, razzledazzle, Robin
29	Предостережения Котлина	Grigory Konushev, Spidfire
30	регистрация в котлин	Konrad Jamrozik, olivierlemasle, oshai
31	сопрограммы	Jemo Mgebrishvili
32	Строительство DSL	Dmitriy L, ice1000
33	Струны	Januson, Sam
34	Тип псевдонимов	Kevin Robatel
35	Тип-безопасные строители	Slav
36	Условные	Abdullah, Alex Facciorusso, jpmcosta, Kirill Rakhman, Robin,

	заявления	Spidfire
37	функции	Aaron Christiansen, baha, Caelum, glee8e, Jayson Minard, KeksArmee, madhead, Spidfire
38	Эквиваленты потока Java 8	Brad, Gerson, Jayson Minard, Piero Divasto, Sam