Руководство по языку Kotlin

Здесь мы собираем ресурсы по Котлину и переводим документацию. Сообщество открыто для новых участников - любого кто может переводить и проверять перевод. Редактирование текста происходит похожим на википедию образом, с той лишь разницей, что тексты и структура меню хранятся в GIT.

Перевод текста дело нехитрое, но качественный перевод котлиновской доки требует хорошего понимания предмета и умения ясно выражать понятое. Это кропотливый творческий процесс. Поэтому, пожалуйста, относитесь с уважением к работе других участников. Но смело вносите изменения, если уверены, что можно что-то улучшить.

Источники

- Официальный сайт языка Kotlin (англ.)
- Исходный код компилятора на GitHub
- Исходники англ. документации
- Раздел на reddit.com, посвященный языку Kotlin

Сообщество

- @KotlinLangRu - Telegram чат посвященный языку Kotlin и переводу документации
- @kotlin_lang - Сообщество разработчиков на Kotlin

Локальные группы:

- @KotlinMoscow - Московская группа
- @KotlinKrasnodar - Краснодарская группа

Актуальные задачи

- Перевести статьи из радела "Java Interop": Calling Java from Kotlin, Calling Kotlin from Java
- Перевести статьи из радела "FAQ": FAQ
- Перевести статьи из раздела "Tools": KDoc, Kapt, Gradle, Maven, Ant, OSGi, Compiler Plugins
- Пометить **ключевые слова** конструкцией `<b class="keyword">keyword-here</b>`

Основной синтаксис

Определение имени пакета

Имя пакета указывается в начале исходного файла, так же как и в Java:

```
package my.demo


import java.util.*


// ...
```

Но в отличие от Java, нет необходимости, чтобы структура пакетов совпадала со структурой папок: исходные файлы могут располагаться в произвольном месте на диске.

См. Пакеты.

Объявление функции

Функция принимает два аргумента `Int` и возвращает `Int`:

```
fun sum(a: Int, b: Int): Int {
  return a + b
}
```

Функция с выражением в качестве тела и автоматически выведенным типом возвращаемого значения:

```
fun sum(a: Int, b: Int) = a + b
```

Функция, не возвращающая никакого значения (void в Java):

```
fun printSum(a: Int, b: Int): Unit {
  print(a + b)
}
```

Тип возвращаемого значения `Unit` может быть опущен:

```
fun printSum(a: Int, b: Int) {
  print(a + b)
}
```

См. Функции.

Определение внутренних переменных

Неизменяемая (только для чтения) внутренняя переменная:

```
val a: Int = 1
val b = 1   // Тип `Int` выведен автоматически
val c: Int  // Тип обязателен, когда значение не инициализируется
```

```
c = 1       // последующее присвоение
```

Изменяемая переменная:

```
var x = 5 // Тип `Int` выведен автоматически
x += 1
```

См. Свойства и поля.

Комментарии

Также, как Java и JavaScript, Kotlin поддерживает однострочные комментарии.

```
// однострочный комментарий


/* Блочный комментарий
   из нескольких строк. */
```

В отличие от Java, блочные комментарии не могут быть вложенными.

См. Documenting Kotlin Code для информации о документации в комментариях.

Использование строковых шаблонов

Допустимо использование переменных внутри строк в формате `$name` или `${name}`:

```
fun main(args: Array<String>) {
  if (args.size == 0) return


  print("Первый аргумент: ${args[0]}")
}
```

См. Строковые шаблоны.

Использование условных выражений

```
fun max(a: Int, b: Int): Int {
  if (a > b)
    return a
  else
    return b
}
```

Также **if** может быть использовано как выражение (т. е. **if** ... **else** возвращает значение):

```
fun max(a: Int, b: Int) = if (a > b) a else b
```

См. Выражение **if**.

Nullable-значения и проверка на **null**

Ссылка должна быть явно объявлена как nullable (символ `?`) когда она может принимать значение **null**.

Возвращает **null** если `str` не содержит числа:

```
fun parseInt(str: String): Int? {
  // ...
}
```

Использование функции, возвращающей **null**:

```
fun main(args: Array<String>) {
  if (args.size < 2) {
    print("Ожидается два целых числа")
    return
  }

  val x = parseInt(args[0])
  val y = parseInt(args[1])

  // Использование `x * y` приведет к ошибке, потому что они могут содержать null
  if (x != null && y != null) {
    // x и y автоматически приведены к не-nullable после проверки на null
    print(x * y)
  }
}
```

или

```
  // ...
  if (x == null) {
    print("Неверный формат числа y '${args[0]}'")
```

```
    return
  }
  if (y == null) {
    print("Неверный формат числа y '${args[1]}'")
    return
  }

  // x и y автоматически приведены к не-nullable после проверки на null
  print(x * y)
```

См. Null-безопасность.

Проверка типа и автоматическое приведение типов

Оператор **is** проверяет, является ли выражение экземпляром заданного типа. Если неизменяемая внутренняя переменная или свойство уже проверены на определенный тип, то в дальнейшем нет необходимости явно приводить к этому типу:

```
fun getStringLength(obj: Any): Int? {
  if (obj is String) {
    // в этом блоке `obj` автоматически преобразован в `String`
    return obj.length
  }

  // `obj` имеет тип `Any` вне блока проверки типа
  return null
}
```

или

```
fun getStringLength(obj: Any): Int? {
  if (obj !is String)
    return null

  // в этом блоке `obj` автоматически преобразован в `String`
  return obj.length
```

```
}
```

или даже

```
fun getStringLength(obj: Any): Int? {
  // `obj` автоматически преобразован в `String` справа от оператора `&&`
  if (obj is String && obj.length > 0)
    return obj.length


  return null
}
```

См. Классы и Приведение типов.

Использование цикла **for**

```
fun main(args: Array<String>) {
  for (arg in args)
    print(arg)
}
```

или

```
for (i in args.indices)
  print(args[i])
```

См. цикл for.

Использование цикла **while**

```
fun main(args: Array<String>) {
  var i = 0
  while (i < args.size)
    print(args[i++])
}
```

См. цикл while.

Использование выражения **when**

```
fun cases(obj: Any) {
```

```
  when (obj) {
    1           -> print("One")
    "Hello"     -> print("Greeting")
    is Long     -> print("Long")
    !is String -> print("Not a string")
    else        -> print("Unknown")
  }
}
```

См. выражение when.

Использование интервалов

Проверка на вхождение числа в интервал с помощью оператора **in**:

```
if (x in 1..y-1)
  print("OK")
```

Проверка значения на выход за пределы интервала:

```
if (x !in 0..array.lastIndex)
  print("Out")
```

Итерация по интервалу:

```
for (x in 1..5)
  print(x)
```

Или по арифметической прогрессии:

```
for (x in 1..10 step 2) {
  print(x)
}
for (x in 9 downTo 0 step 3) {
  print(x)
}
```

См. Интервалы.

Использование коллекций

Итерация по коллекции:

```
for (name in names)
  println(name)
```

Проверка, содержит ли коллекция данный объект, с помощью оператора **in**:

```
val items = setOf("apple", "banana", "kiwi")
when {
  "orange" in items -> println("juicy")
  "apple" in items -> println("apple is fine too")
}
```

Использование лямбда-выражения для фильтрации и модификации коллекции:

```
names
    .filter { it.startsWith("A") }
    .sortedBy { it }
    .map { it.toUpperCase() }
    .forEach { print(it) }
```

См. Функции высшего порядка и лямбды.

Идиомы

Набор различных часто используемых идиом в языке Kotlin. Если у вас есть любимая идиома, вы можете поделится ею здесь. Для этого нужно выполнить pull request.

Создание DTO (он же POJO или POCO)

```
data class Customer(val name: String, val email: String)
```

создаёт класс `Customer`, обладающий следующими возможностями:

- геттеры (и сеттеры в случае **var**'s) для всех свойств
- метод `equals()`
- метод `hashCode()`
- метод `toString()`
- метод `copy()`
- методы `component1()`, `component2()`, и т.п. для всех свойств (см. Классы данных)

Значения по умолчанию для параметров функций

```
fun foo(a: Int = 0, b: String = "") { ... }
```

Фильтрация списка

```
val positives = list.filter { x -> x > 0 }
```

Или короче:

```
val positives = list.filter { it > 0 }
```

Форматирование строк

```
println("Name $name")
```

Проверка объекта на принадлежность к определённому классу

```
when (x) {
    is Foo -> ...
    is Bar -> ...
    else   -> ...
}
```

Итерация по карте/списку пар

```
for ((k, v) in map) {
    println("$k -> $v")
}
```

Имена переменных `k` и `v` не имеют значения

Использование последовательностей чисел

```
for (i in 1..100) { ... }
for (x in 2..10) { ... }
```

Read-only список

```
val list = listOf("a", "b", "c")
```

Read-only ассоциативный список (map)

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

Обращение к ассоциативному списку

```
println(map["key"])

map["key"] = value
```

Ленивые свойства

```
val p: String by lazy {

    // compute the string

}
```

Функции-расширения

```
fun String.spaceToCamelCase() { ... }


"Convert this to camelcase".spaceToCamelCase()
```

Создание синглтона

```
object Resource {

    val name = "Name"

}
```

Сокращение для "Если не null"

```
val files = File("Test").listFiles()


println(files?.size)
```

Сокращение для "Если не null, иначе"

```
val files = File("Test").listFiles()


println(files?.size ?: "empty")
```

Вызов оператора при равенстве null

```
val data = ...

val email = data["email"] ?: throw IllegalStateException("Email is missing
!")
```

Выполнение при неравенстве null

```
val data = ...


data?.let {
    ... // execute this block if not null
}
```

Return с оператором when

```
fun transform(color: String): Int {
    return when (color) {
        "Red" -> 0
        "Green" -> 1
        "Blue" -> 2
        else -> throw IllegalArgumentException("Invalid color param value"
)
    }
}
```

'try/catch' как выражение

```
fun test() {
    val result = try {
        count()
    } catch (e: ArithmeticException) {
        throw IllegalStateException(e)
    }


    // Working with result
}
```

'if' как выражение

```
fun foo(param: Int) {
    val result = if (param == 1) {
        "one"
```

```
    } else if (param == 2) {
        "two"
    } else {
        "three"
    }
}
```

Builder-style использование методов, возвращающих `Unit`

```
fun arrayOfMinusOnes(size: Int): IntArray {
    return IntArray(size).apply { fill(-1) }
}
```

Функции, состоящие из одного выражения

```
fun theAnswer() = 42
```

Что равносильно этому:

```
fun theAnswer(): Int {
    return 42
}
```

Для сокращения кода их можно эффективно совмещать с другими идиомами. Например с **when**:

```
fun transform(color: String): Int = when (color) {
    "Red" -> 0
    "Green" -> 1
    "Blue" -> 2
    else -> throw IllegalArgumentException("Invalid color param value")
}
```

Вызов нескольких методов объекта ('with')

```
class Turtle {
    fun penDown()
    fun penUp()
    fun turn(degrees: Double)
```

```
    fun forward(pixels: Double)
}


val myTurtle = Turtle()
with(myTurtle) { //draw a 100 pix square
    penDown()
    for(i in 1..4) {
        forward(100.0)
        turn(90.0)
    }
    penUp()
}
```

try with resources из Java 7

```
val stream = Files.newInputStream(Paths.get("/some/file.txt"))
stream.buffered().reader().use { reader ->
    println(reader.readText())
}
```

Соглашение о стилистике кода

Данная страница содержит описание текущего стиля написания кода на языке Kotlin.

Правила наименований

При возникновении сомнений по умолчанию используются следующие правила:

- используйте camelCase в названиях (а также избегайте подчёркиваний)
- названия типов пишутся с заглавной буквы
- названия методов и свойств начинаются со строчной буквы
- используйте отступ из 4 пробелов
- функции, объявленные как public, желательно снабжать документацией в стиле документации языка Kotlin

Двоеточие

В тех случаях, когда двоеточие разделяет тип и подтип, перед двоеточием ставится пробел. Если же двоеточие ставится между сущностью и типом, то пробел опускается:

```
interface Foo<out T : Any> : Bar {
    fun foo(a: Int): T
}
```

Лямбда-выражения

В лямбда-выражениях фигурные скобки, а также стрелка и параметры отделяются пробелами. Желательно передавать лямбду за пределами скобок.

```
list.filter { it > 10 }.map { element -> element * 2 }
```

В коротких лямбда-выражениях, не являющихся вложенными, рекомендуется использовать соглашение `it` вместо явного объявления параметра. Во вложенных лямбдах с параметрами последние всегда должны быть объявлены.

Объявление классов

Классы с небольшим количеством аргументов можно писать на одной строчке:

```
class Person(id: Int, name: String)
```

Классы с более длинными сигнатурами должны быть отформатированны так, чтобы каждый параметр располагался с новой строки

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name) {
    // ...
}
```

Если класс расширяет несколько интерфейсов, конструктор суперкласса (если он есть) должен располагаться на первой строке, а после него, список расширяемых интерфейсов: каждый интерфейс с новой строки.

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name),
    KotlinMaker {
```

```
    // ...
}
```

Для параметров конструктора может использоваться как обычный отступ, так и двойной.

Тип Unit

Если возвращаемым типом является Unit, то его можно явно не указывать:

```
fun foo() { // здесь пропущено ": Unit"


}
```

Функции vs Свойства

В некоторых случаях, функции без аргументов могут быть взаимозаменяемы с неизменяемыми (read-only) свойствами. Несмотря на схожую семантику, есть некоторые стилистические соглашения, указывающие на то, когда лучше использовать одно из этих решений.

Использование свойства перед функцией предпочтительнее, когда лежащий в основе алгоритм:

- не выбрасывает исключений
- имеет `O(1)` сложность
- не требует больших затрат на выполнение (или результат вычислений кэшируется при первом вызове)
- возвращает одинаковый результат

Основные типы

В Kotlin всё является объектом, в том смысле, что пользователь может вызвать функцию или получить доступ к свойству любой переменной. Некоторые типы являются встроенными, т.к. их реализация оптимизирована, хотя для пользователя они выглядеть как обычные классы. В данном разделе описывается большинство этих типов: числа, символы, логические переменные и массивы.

Числа

Kotlin обрабатывает численные типы примерно так же, как и Java, хотя некоторые различия всё же присутствуют. Например, отсутствует неявное расширяющее преобразование для чисел, а литералы в некоторых случаях немного отличаются.

Для представления чисел в Kotlin используются следующие встроенные типы (подобные типам в Java):

Тип	Количество бит
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

Обратите внимание, что символы (characters) в языке Kotlin не являются числами (в отличие от Java).

Символьные постоянные

В языке Kotlin присутствуют следующие виды символьных постоянных (констант) для целых значений:

- Десятичные числа: `123`
  - Тип Long обозначается заглавной `L`: `123L`
- Шестнадцатеричные числа: `0x0F`
- Двоичные числа: `0b00001011`

ВНИМАНИЕ: Восьмеричные литералы не поддерживаются.

Также Kotlin поддерживает числа с плавающей запятой:

- Тип Double по умолчанию: `123.5`, `123.5e10`
- Тип Float обозначается с помощью `f` или `F`: `123.5f`

Представление

Обычно платформа Java хранит числа в виде примитивных типов JVM; если же нам необходима ссылка, которая может принимать значение null (например, `Int?`), то используются обёртки. В приведённом ниже примере показано использование обёрток.

Обратите внимание, что использование обёрток для одного и того же числа не гарантирует равенства ссылок на них:

```
val a: Int = 10000

print(a === a) // Prints 'true'

val boxedA: Int? = a

val anotherBoxedA: Int? = a

print(boxedA === anotherBoxedA) // !!!Prints 'false'!!!
```

Однако, равенство по значению сохраняется:

```
val a: Int = 10000

print(a == a) // Prints 'true'

val boxedA: Int? = a

val anotherBoxedA: Int? = a

print(boxedA == anotherBoxedA) // Prints 'true'
```

Явные преобразования

Из-за разницы в представлениях меньшие типы не являются подтипами бОльших типов. В противном случае у нас возникли бы сложности:

```
// Возможный код, который на самом деле не скомпилируется:

val a: Int? = 1 // "Обёрнутый" Int (java.lang.Integer)

val b: Long? = a // неявное преобразование возвращает "обёрнутый" Long (java.lang.Long)

print(a == b) // Нежданчик! Данное выражение выведет "false" т. к. метод equals() типа Long предполагает, что вторая часть выражения также имеет тип Long
```

Таким образом, будет утрачена не только тождественность (равенство по ссылке), но и равенство по значению.

Как следствие, неявное преобразование меньших типов в большие НЕ происходит. Это значит, что мы не можем присвоить значение типа `Byte`переменной типа `Int` без явного преобразования:

```
val b: Byte = 1 // порядок, литералы проверяются статически

val i: Int = b // ОШИБКА
```

Мы можем использовать явное преобразование для "сужения" чисел

```
val i: Int = b.toInt() // порядок: число явно сужено
```

Каждый численный тип поддерживает следующие преобразования:

- `toByte(): Byte`
- `toShort(): Short`
- `toInt(): Int`

- `toLong(): Long`
- `toFloat(): Float`
- `toDouble(): Double`
- `toChar(): Char`

Отсутствие неявного преобразования редко бросается в глаза, поскольку тип выводится из контекста, а арифметические действия перегружаются для подходящих преобразований, например:

```
val l = 1L + 3 // Long + Int => Long
```

Арифметические действия

Kotlin поддерживает обычный набор арифметических действий над числами, которые объявлены членами соответствующего класса (тем не менее, компилятор оптимизирует вызовы вплоть до соответствующих инструкций). См. Перегрузка операторов.

Что касается битовых операций, то вместо особых обозначений для них используются именованные функции, которые могут быть вызваны в инфиксной форме, к примеру:

```
val x = (1 shl 2) and 0x000FF000
```

Ниже приведён полный список битовых операций (доступны только для типов `Int` и `Long`):

- `shl(bits)` – сдвиг влево с учётом знака (`<<` в Java)
- `shr(bits)` – сдвиг вправо с учётом знака (`>>` в Java)
- `ushr(bits)` – сдвиг вправо без учёта знака (`>>>` в Java)
- `and(bits)` – побитовое И
- `or(bits)` – побитовое ИЛИ
- `xor(bits)` – побитовое исключающее ИЛИ
- `inv()` – побитовое отрицание

Символы

Символы в Kotlin представлены типом `Char`. Напрямую они не могут рассматриваться в качестве чисел

```
fun check(c: Char) {
  if (c == 1) { // ОШИБКА: несовместимый тип
    // ...
  }
}
```

Символьные литералы записываются в одинарных кавычках: `'1'`, `'\n'`, `'\uFF00'`. Мы можем явно привести символ в число типа `Int`

```
fun decimalDigitValue(c: Char): Int {
  if (c !in '0'..'9')
```

```
    throw IllegalArgumentException("Вне диапазона")
  return c.toInt() - '0'.toInt() // Явные преобразования в число
}
```

Подобно числам, символы оборачиваются при необходимости использования nullable ссылки. При использовании обёрток тождественность (равенство по ссылке) не сохраняется.

Логический тип

Тип `Boolean` представляет логический тип данных и принимает два значения: **true** и **false**.

При необходимости использования nullable ссылок логические переменные оборачиваются.

Встроенные действия над логическими переменными включают

- `||` – ленивое логическое ИЛИ
- && – ленивое логическое И
- `!` - отрицание

Массивы

Массивы в Kotlin представлены классом `Array`, обладающим функциями `get` и `set` (которые обозначаются `[]` согласно соглашению о перегрузке операторов), и свойством `size`, а также несколькими полезными встроенными функциями:

```
class Array<T> private constructor() {
  val size: Int
  fun get(index: Int): T
  fun set(index: Int, value: T): Unit


  fun iterator(): Iterator<T>
  // ...
}
```

Для создания массива мы можем использовать библиотечную функцию `arrayOf()`, которой в качестве аргумента передаются элементы массива, т.е. выполнение `arrayOf(1, 2, 3)` создаёт массив [1, 2, 3]. С другой стороны библиотечная функция `arrayOfNulls()` может быть использована для создания массива заданного размера, заполненного значениями null.

Также для создания массива можно использовать фабричную функцию, которая принимает размер массива и функцию, возвращающую начальное значение каждого элемента по его индексу:

```
// создаёт массив типа Array<String> со значениями ["0", "1", "4", "9", "16"]

val asc = Array(5, { i -> (i * i).toString() })
```

Как отмечено выше, оператор `[]` используется вместо вызовов встроенных функций `get()` и `set()`.

Обратите внимание: в отличие от Java массивы в Kotlin являются неизменяемыми. Это значит, что Kotlin запрещает нам присваивать значение `Array<String>` массиву типа `Array<Any>`, предотвращая таким образом возможный отказ во время исполнения (хотя вы можете использовать `Array<out Any>`, см. Type Projections).

Также в Kotlin есть особые классы для представления массивов примитивных типов без дополнительных затрат на оборачивание: `ByteArray`, `ShortArray`, `IntArray` и т.д. Данные классы не наследуют класс `Array`, хотя и обладают тем же набором методов и свойств. У каждого из них есть соответствующая фабричная функция:

```
val x: IntArray = intArrayOf(1, 2, 3)

x[0] = x[1] + x[2]
```

Строки

Строки в Kotlin представлены типом `String`. Строки являются неизменяемыми. Строки состоят из символов, которые могут быть получены по порядковому номеру: `s[i]`. Проход по строке выполняется циклом **for**:

```
for (c in str) {

  println(c)

}
```

Строковые литералы

В Kotlin представлены два типа строковых литералов: строки с экранированными символами и обычные строки, могущие содержать символы новой строки и произвольный текст. Экранированная строка очень похожа на строку в Java:

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

Экранирование выполняется общепринятым способом, а именно с помощью обратной косой черты.

Обычная строка выделена тройной кавычкой (`"""`), не содержит экранированных символов, но может содержать символы новой строки и любые другие символы:

```
val text = """
```

```
  for (c in "foo")
    print(c)
"""
```

Строковые шаблоны

Строки могут содержать шаблонные выражения, т.е. участки кода, которые выполняются, а полученный результат встраивается в строку. Шаблон начинается со знака доллара ($) и состоит либо из простого имени (например, переменной):

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

либо из произвольного выражения в фигурных скобках:

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

Шаблоны поддерживаются как в обычных, так и в экранированных строках. При необходимости символ `$` может быть представлен с помощью следующего синтаксиса:

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

Пакеты

Файл с исходным кодом может начинаться с объявления пакета:

```
package foo.bar

fun baz() {}

class Goo {}

// ...
```

Всё содержимое файла с исходниками (например, классы и функции) располагается в объявленном пакете. Таким образом, в приведённом выше примере полное имя функции `baz()` будет `foo.bar.baz`, а полное имя класса `Goo` - `foo.bar.Goo`.

Если файл не содержит явного объявления пакета, то его содержимое находится в безымянном "пакете по умолчанию".

Импорт

Помимо импорта по умолчанию каждый файл может содержать свои собственные объявления импорта. Синтаксис импорта описан а разделе Грамматика.

Мы можем импортировать одно имя, например

```
import foo.Bar // теперь Bar можно использовать без указания пакета
```

или доступное содержимое пространства имён (пакет, класс, объект и т.д.):

```
import foo.* // всё в 'foo' становится доступно без указания пакета
```

При совпадении имён мы можем разрешить коллизию используя ключевое слово **as** для локального переименования совпадающей сущности:

```
import foo.Bar // Bar доступен

import bar.Bar as bBar // bBar заменяет имя 'bar.Bar'
```

Ключевое слово `import` можно использовать не только с классами, но и с другими объявлениями:

- функции и свойства верхнего уровня;
- функции и свойства, объявленные в объявлениях объектов;
- перечислениях

В отличие от Java, Kotlin не предоставляет отдельного объявления статического импорта "import static"; все подобные объявления импортируются ключевым словом `import`.

Область видимости объявлений верхнего уровня

Если объявление верхнего уровня отмечено как **private**, то оно является private в файле, в котором оно объявлено (см. Модификаторы области видимости).

Управляющие инструкции

Условное выражение **if**

В языке Kotlin ключевое слово **if** является выражением, т.е. оно возвращает значение. Это позволяет отказаться от тернарного оператора (условие ? условие истинно : условие ложно), поскольку выражению **if** вполне по силам его заменить.

```
// обычное использование

var max = a

if (a < b)

  max = b
```

```
// с блоком else
var max: Int
if (a > b)
  max = a
else
  max = b


// в виде выражения
val max = if (a > b) a else b
```

"Ветви" выражения **if** могут содержать несколько строк кода, при этом последнее выражение является значением блока:

```
val max = if (a > b) {
    print("возвращаем a")
    a
  }
  else {
    print("возвращаем b")
    b
  }
```

Если вы используете конструкцию **if** в качестве выражения (например, возвращая его значение или присваивая его переменной), то использование ветки `else` является обязательным.

См. использование **if**.

Условное выражение **when**

Ключевое слово **when** призвано заменить оператор switch, присутствующий в C-подобных языках. В простейшем виде его использование выглядит так:

```
when (x) {
  1 -> print("x == 1")
  2 -> print("x == 2")
  else -> { // обратите внимание на блок
    print("x is neither 1 nor 2")
```

```
    }
}
```

Выражение **when** последовательно сравнивает аргумент со всеми указанными значениями до удовлетворения одного из условий. **when** можно использовать и как выражение, и как оператор. При использовании в виде выражения значение ветки, удовлетворяющей условию, становится значением всего выражения. При использовании в виде оператора значения отдельных веток отбрасываются. (В точности как **if**: каждая ветвь может быть блоком и её значением является значение последнего выражения блока.)

Значение ветки **else** вычисляется в том случае, когда ни одно из условий в других ветках не удовлетворено. Если **when** используется как выражение, то ветка **else** является обязательной, за исключением случаев, в которых компилятор может убедиться, что ветки покрывают все возможные значения.

Если для нескольких значений выполняется одно и то же действие, то условия можно перечислять в одной ветке через запятую:

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

Помимо констант в ветках можно использовать произвольные выражения:

```
when (x) {
    parseInt(s) -> print("s encodes x")
    else -> print("s does not encode x")
}
```

Также можно проверять вхождение аргумента в интервал **in** или **!in** или его наличие в коллекции:

```
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
}
```

Помимо этого Kotlin позволяет с момощью **is** или **!is** проверить тип аргумента. Обратите внимание, что благодаря умным приведениям вы можете получить доступ к методам и свойствам типа без дополнительной проверки:

```
val hasPrefix = when(x) {
```

```
  is String -> x.startsWith("prefix")
  else -> false
}
```

when удобно использовать вместо цепочки условий вида **if**-**else if**. При отстутствии аргумента условия работают как простые логические выражения, а тело ветки выполняется при его истинности:

```
when {
  x.isOdd() -> print("x is odd")
  x.isEven() -> print("x is even")
  else -> print("x is funny")
}
```

См. использование **when**.

Циклы **for**

Цикл **for** обеспечивает перебор всех значений, поставляемых итератором. Для этого используется следующий синтаксис:

```
for (item in collection)
  print(item)
```

Телом цикла может быть блок кода:

```
for (item: Int in ints) {
  // ...
}
```

Как отмечено выше, цикл **for** позволяет проходить по всем элементам объекта, имеющего итератор, например:

- обладающего внутренней или внешней функцией `iterator()`, возвращаемый тип которой
  - обладает внутренней или внешней функцией `next()`, и
  - обладает внутренней или внешней функцией `hasNext()`, возвращающей `Boolean`.

Все три указанные функции должны быть объявлены как `operator`.

Если при проходе по массиву или списку необходим порядковый номер элемента, используйте следующий подход:

```
for (i in array.indices)
  print(array[i])
```

Обратите внимание, что данная "итерация по ряду" компилируется в более производительный код без создания дополнительных объектов.

Также вы можете использовать библиотечную функцию `withIndex`:

```
for ((index, value) in array.withIndex()) {
    println("the element at $index is $value")
}
```

См. использование **for**.

Циклы **while**

Ключевые слова **while** и **do**..**while** работают как обычно:

```
while (x > 0) {
  x--
}


do {
  val y = retrieveData()
} while (y != null) // y is visible here!
```

См. использование **while**.

Ключевые слова **break** и **continue** в циклах

Kotlin поддерживает обычные операторы **break** и **continue** в циклах. См. Операторы перехода.

Операторы перехода

В **Kotlin** определено три оператора перехода:

- **return** по умолчанию производит возврат из ближайшей окружающей его функции или анонимной функции
- **break** завершает выполнение цикла
- **continue** продолжает выполнение цикла со следующего его шага, без обработки оставшегося кода текущей итерации

Метки операторов **break** и **continue**

Любое выражение в **Kotlin** может быть помечено меткой **label**. Метки имеют идентификатор в виде знака `@`. Например: метки `abc@`, `fooBar@` являются корректными (см. грамматика). Для того, чтобы пометить выражение, мы просто ставим метку перед ним:

```
loop@ for (i in 1..100) {
  // ...
}
```

Теперь мы можем уточнить значения операторов **break** или **continue** с помощью меток:

```
loop@ for (i in 1..100) {
  for (j in 1..100) {
    if (...)
      break@loop
  }
}
```

Оператор **break**, отмеченный `@loop`, переводит выполнение кода к той его части, которая находится сразу после соответствующей метки `loop@`.
Оператор **continue** продолжает цикл со следующей его итерации.

Возврат к меткам

В **Kotlin** функции могут быть вложены друг в друга с помощью анонимных объектов, локальных функций (ориг.:*local functions*) и *function literals*. Подходящий **return** позволит вернуться из внешней функции. Одним из самых удачных применений этой синтаксической конструкции служит возврат из лямбда-выражения. Подумайте над этим утверждением, читая данный пример:

```
fun foo() {
  ints.forEach {
    if (it == 0) return
    print(it)
  }
}
```

Опертор **return** возвращается из ближаейшей функции, в нашем случае `foo`. (Обратите внимание, что такой местный возврат поддерживается только лямбда-выражениями, переданными инлайн-функциям.) Если нам надо вернуться из лямбда-выражения, к

оператору стоит поставить метку и тем самым сделать уточнение для ключевого слова **return**:

```
fun foo() {
  ints.forEach lit@ {
    if (it == 0) return@lit
    print(it)
  }
}
```

Теперь он возвращается только из лямда-выражения. Зачастую намного более удобно указывать метки неявно: такие метки имеют такое же имя, как и функция, к которой относится лямбда.

```
fun foo() {
  ints.forEach {
    if (it == 0) return@forEach
    print(it)
  }
}
```

Возможно также использование анонимной функции в качестве альтернативы лямбда-выражениям. Оператор **return** вовзращается из самой анонимной функции.

```
fun foo() {
  ints.forEach(fun(value: Int) {
    if (value == 0) return
    print(value)
  })
}
```

При возвращении значения парсер отдаёт предпочтение специализированному возврату, типа

```
return@a 1
```

что значит "верни `1` в метке `@a`, а не "верни выражение с меткой `(@a 1)`".

Классы и наследование

Классы

Классы в **Kotlin** объявляются с помощью использования ключевого слова **class**:

```
class Invoice {

}
```

Объявление класса состоит из имени класса, заголовка (указания типов его параметров, первичного конструктора и т.п) и тела класса, заключённого в фигурные скобки. И заголовок, и тело класса являются необязательными составляющими: если у класса нет тела, фигурные скобки могут быть опущены.

```
class Empty
```

Конструкторы

Класс в **Kotlin** может иметь первичный конструктор (**primary constructor**) и один или более вторичных конструкторов (**secondary constructors**). Первичный конструктор является частью заголовка класса, его объявление идёт сразу после имени класса (и необязательных параметров):

```
class Person constructor(firstName: String)
```

Если у конструктора нет аннотаций и модификаторов видимости, ключевое слово **constructor** может быть опущено:

```
class Person(firstName: String)
```

Первичный конструктор не может содержать в себе исполняемого кода. Инициализирующий код может быть помещён в соответствующий блок (**initializers blocks**), который помечается словом **init**:

```
class Customer(name: String) {

    init {

        logger.info("Customer initialized with value ${name}")

    }

}
```

Обратите внимание, что параметры первичного конструктора могут быть использованы в инициализирующем блоке. Они также могут быть использованы при инициализации свойств в теле класса:

```
class Customer(name: String) {

    val customerKey = name.toUpperCase()

}
```

В действительности, для объявления и инициализации свойств первичного конструктора в **Kotlin** есть лаконичное синтаксическое решение:

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

  // ...
```

```
}
```

Свойства, объявленные в первичном конструкторе, могут быть изменяемые (**var**) и неизменяемые (**val**).

Если у конструктора есть аннотации или модификаторы видимости, ключевое слово **constructor**обязательно, и модификаторы используются перед ним:

```
class Customer public @Inject constructor(name: String) { ... }
```

Для более подробной информации по данному вопросу см. "Модификаторы доступа".

Второстепенные конструкторы

В классах также могут быть объявлены дополнительные конструкторы (**secondary constructors**), перед которыми используется ключевое слово **constructor**:

```
class Person {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

Если у класса есть главный (первичный) конструктор, каждый последующий конструктор должен прямо или косвенно ссылаться (через другой(*ие*) конструктор(*ы*)) на первичный:

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Если в абстрактном классе не объявлено никаких конструкторов (первичных или второстепенных), у этого класса автоматически сгенерируется пустой конструктор без параметров. Видимость этого конструктора будет **public**. Если вы не желаете иметь класс с открытым **public** конструктором, вам необходимо объявить пустой конструктор с соответствующим модификатором видимости:

```
class DontCreateMe private constructor () {
}
```

Примечание: В виртуальной машине JVM компилятор генерирует дополнительный конструктор без параметров в случае, если все параметры первичного конструктора имеют значения по умолчанию. Это делает использование таких библиотек, как **Jackson** и **JPA**, более простым в языке **Kotlin**, так как они используют пустые конструкторы при создании экземпляров классов.

```
class Customer(val customerName: String = "")
```

Создание экземпляров классов

Для создания экземпляра класса конструктор вызывается так, как если бы он был обычной функцией:

```
val invoice = Invoice()


val customer = Customer("Joe Smith")
```

Обращаем ваше внимание на то, что в **Kotlin** не используется ключевое слово **new**.

Члены класса

Классы могут содержать в себе:

- Конструкторы и инициализирующие блоки
- Функции
- Свойства
- Вложенные классы
- Объявления объектов

Наследование

Для всех классов в языке **Koltin** родительским суперклассом является класс `Any`. Он также является родительским классом для любого класса, в котором не указан какой-либо другой родительский класс:

```
class Example // Implicitly inherits from Any
```

Класс `Any` не является аналогом `java.lang.Object`. В частности, у него нет никаких членов кроме методов: `equals()`, `hashCode()`, и `toString()`. Пожалуйста, ознакомьтесь с совместимостью с Javaдля более подробной информации.

Для явного объявления суперкласса мы помещаем его имя за знаком двоеточия в оглавлении класса:

```
open class Base(p: Int)


class Derived(p: Int) : Base(p)
```

Если у класса есть основной конструктор, базовый тип может (и должен) быть проинициализирован там же, с использованием параметров первичного конструктора.

Если у класса нет первичного конструктора, тогда каждый последующий второстепенный конструктор должен включать в себя инициализацию базового типа с помощью ключевого слова **super** или давать отсылку на другой конструктор, который это делает.

Примечательно, что любые вторичные конструкторы могут ссылаться на разные конструкторы базового типа:

```
class MyView : View {
    constructor(ctx: Context) : super(ctx) {
    }

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs) {
    }
}
```

Ключевое слово **open** является противоположностью слову **final** в **Java**: оно позволяет другим классам наследоваться от данного. По умолчанию, все классы в **Kotlin** имеют статус **final**, что отвечает Effective Java, Item 17: *Design and document for inheritance or else prohibit it*.

Переопределение членов класса

Как упоминалось ранее, мы придерживаемся идеи определённости и ясности в языке **Kotlin**. И, в отличие от **Java**, **Kotlin** требует чёткой аннотации и для членов, которые могут быть переопределены, и для самого переопределения:

```
open class Base {
  open fun v() {}
  fun nv() {}
}
class Derived() : Base() {
  override fun v() {}
}
```

Для `Derived.v()` необходима аннотация **override**. В случае её отсутствия компилятор выдаст ошибку. Если у функции типа `Base.nv()` нет аннотации **open**, объявление метода с такой же сигнатурой в производном классе невозможно, с **override** или без.
В **final** классе (классе без аннотации **open**), запрещено использование аннотации **open** для его членов.

Член класса, помеченный **override**, является сам по себе **open**, т.е. он может быть переопределён в производных классах. Если вы хотите запретить возможность переопределения такого члена, используйте **final**:

```
open class AnotherDerived() : Base() {
  final override fun v() {}
}
```

Стойте! Как мне теперь хакнуть свои библиотеки?

При нашем подходе к переопределению классов и их членов (которые по дефолту **final**) будет сложно унаследоваться от чего-нибудь внутри используемых вами библиотек для того, чтобы переопределить не предназначенный для этого метод и внедрить туда свой гнусный хак.

Мы думаем, что это не является недостатком по следующим причинам:

- Опыт поколений говорит о том, что, в любом случае, лучше не позволять внедрять такие хаки
- Люди успешно используют другие языки (**C++**, **C#**), которые имеют аналогичных подход к этому вопросу
- Если кто-то действительно хочет хакнуть, пусть напишет свой код на **Java** и вызовет его в **Kotlin** *(см. Java-совместимость)*

Правила переопределения

В **Kotlin** правила наследования имплементации определены следующим образом: если класс перенимает большое количество имплементаций одного и того члена от ближайших родительских классов, он должен переопределить этот член и обеспечить свою собственную имплементацию (возможно, используя одну из унаследованных). Для того, чтобы отметить супертип (родительский класс), от которого мы унаследовали данную имплементацию, мы используем ключевое слово **super**. Для уточнения имя родительского супертипа используются треугольные скобки, например `super<Base>`:

```
open class A {
  open fun f() { print("A") }
  fun a() { print("a") }
}

interface B {
  fun f() { print("B") } // interface members are 'open' by default
  fun b() { print("b") }
}

class C() : A(), B {
  // The compiler requires f() to be overridden:
  override fun f() {
    super<A>.f() // call to A.f()
    super<B>.f() // call to B.f()
```

```
  }
}
```

Нормально наследоваться одновременно от `A` и `B`. У нас не возникнет никаких проблем с `a()` и `b()` в том случае, если `C` унаследует только одну имплементацию этих функций. Но для `f()` у нас есть две имплементации, унаследованные классом `C`, поэтому необходимо переопределить `f()` в `C` и обеспечить нашу собственную реализацию этого метода для устранения получившейся неоднозначности.

Абстрактные классы

Класс и некоторые его члены могут быть объявлены как **abstract**. Абстрактный член не имеет реализации в его классе. Обратите внимание, что нам не надо аннотировать абстрактный класс или функцию словом **open** - это подразумевается и так.

Можно переопределить не-абстрактный **open** член абстрактным

```
open class Base {
  open fun f() {}
}


abstract class Derived : Base() {
  override abstract fun f()
}
```

Объекты-помощники

В **Kotlin**, в отличие от **Java** или **C#**, в классах не бывает статических методов. В большинстве случаев рекомендуется использовать функции на уровне пакета (ориг.: *"package-level functions"*).

Если вам нужно написать функцию, которая может быть использована без создания экземпляра класса, но имела бы доступ к данным внутри этого класса (к примеру, фабричный метод), вы можете написать её как член объявления объекта внутри этого класса.

В частности, если вы объявляете объект-помощник в своём классе, у вас появляется возможность обращаться к его членам, используя тот же синтаксис, как при использовании статических методов в **Java**/**C#** (указав название класса для доступа).

Прочие классы

Также обратите внимание на:

- Изолированные классы (sealed classes)

- Классы данных (data classes)
- Вложенные классы (nested classes)
- Классы-перечисления (enum classes)

Свойства и поля

Объявление свойств

Классы в **Kotlin** могут иметь свойства: изменяемые (*mutable*) и неизменяемые (*read-only*) — **var** и **val**соответственно.

```
public class Address {
    public var name: String = ...
    public var street: String = ...
    public var city: String = ...
    public var state: String? = ...
    public var zip: String = ...
}
```

Для того, чтобы воспользоваться свойством, мы просто обращаемся к его имени (как в **Java**):

```
fun copyAddress(address: Address): Address {
    val result = Address() // нет никакого слова `new`
    result.name = address.name // вызов методов доступа
    result.street = address.street
    // ...
    return result
}
```

Геттеры и сеттеры

Полный синтаксис объявления свойства выглядит так:

```
var <propertyName>: <PropertyType> [= <property_initializer>]
    [<getter>]
    [<setter>]
```

Инициализатор *property_initializer*, геттер и сеттер можно не указывать. Также необязательно указывать тип свойства, если он может быть выведен из контекста или наследован от базового класса.

Примеры:

```
var allByDefault: Int? // ошибка: необходима явная инициализация, предусмотрены стандартные геттер и сеттер

var initialized = 1 // имеет тип Int, стандартный геттер и сеттер
```

Синтаксис объявления констант имеет два отличия от синтаксиса объявления изменяемых переменных: во-первых, объявление начинается с ключевого слова `val` вместо `var`, а во-вторых, объявление сеттера запрещено:

```
val simple: Int? // имеет тип Int, стандартный геттер, должен быть инициализирован в конструкторе

val inferredType = 1 // имеет тип Int и стандартный геттер
```

Мы можем самостоятельно описать методы доступа, как и обычные функции, прямо при объявлении свойств. Например, пользовательский геттер:

```
val isEmpty: Boolean

    get() = this.size == 0
```

Пользовательский сеттер выглядит примерно так:

```
var stringRepresentation: String

    get() = this.toString()

    set(value) {

        setDataFromString(value) // парсит строку и устанавливает значения для других свойств

    }
```

По договорённости, имя параметра сеттера - `value`, но вы можете использовать любое другое.

Если вам нужно изменить область видимости метода доступа или пометить его аннотацией, при этом не внося изменения в реализацию по умолчанию, вы можете объявить метод доступа без объявления его тела:

```
var setterVisibility: String = "abc"

    private set // сеттер имеет private доступ и стандартную реализацию



var setterWithAnnotation: Any? = null

    @Inject set // аннотирование сеттера с помощью Inject
```

Backing Fields

Классы в **Kotlin** не могут иметь полей. Т.е. переменные, которые вы объявляете внутри класса только выглядят и ведут себя как поля из Java, хотя на самом деле являются *свойствами*, т.к. для них неявно реализуются методы get и set. А сама переменная, в которой находится значение свойства, называется **backing field**. Однако, иногда, при использовании пользовательских методов доступа, необходимо иметь доступ к *backing field*. Для этих целей **Kotlin** предоставляет автоматическое *backing field*, к которому можно обратиться с помощью идентификатора `field`:

```
var counter = 0

    set(value) {

        if (value >= 0) field = value // значение при инициализации записывается прямиком в backing field

    }
```

Идентификатор `field` может быть использован только в методах доступа к свойству.

Backing field будет сгенерировано для свойства, если оно использует стандартную реализацию как минимум одного из методов доступа. Или в случае, когда пользовательский метод доступа ссылается на него через идентификатор `field`.

Например, в нижестоящем примере не будет никакого *backing field*:

```
val isEmpty: Boolean

    get() = this.size == 0
```

Backing Properties

Если вы хотите предпринять что-то такое, что выходит за рамки вышеуказанной схемы "неявного *backing field*", вы всегда можете использовать *backing property*:

```
private var _table: Map<String, Int>? = null

public val table: Map<String, Int>

    get() {

        if (_table == null) {

            _table = HashMap() // параметры типа вычисляются автоматически (ориг.: "Type parameters are inferred")

        }

        return _table ?: throw AssertionError("Set to null by another thread")

    }
```

Такой подход ничем не отличается от подхода в Java, так как доступ к приватным свойствам со стандартными геттерами и сеттерами оптимизируется таким образом, что вызов функции не происходит.

Константы времени компиляции

Свойства, значение которых известно во время компиляции, могут быть помечены как *константы времени компиляции*. Для этого используется модификатор `const`. Такие свойства должны соответствовать следующим требованиям:

- Находиться на самом высоком уровне или быть членом объекта `object`
- Быть проинициализированными значением типа `String` или значением примитивного типа
- Не иметь переопределённого геттера

Такие свойства могут быть использованы в аннотациях:

```
const val SUBSYSTEM_DEPRECATED: String = "This subsystem is deprecated"


@Deprecated(SUBSYSTEM_DEPRECATED) fun foo() { ... }
```

Свойства с поздней инициализацией

Обычно, свойства, объявленные non-null типом, должны быть проинициализированы в конструкторе. Однако, довольно часто это неосуществимо. К примеру, свойства могут быть инициализированы через внедрение зависимостей, в установочном методе (ориг.: *"setup method"*) юнит-теста или в методе `onCreate` в Android. В таком случае вы не можете обеспечить non-null инициализацию в конструкторе, но всё равно хотите избежать проверок на null при обращении внутри тела класса к такому свойству.

Для того, чтобы справиться с такой задачей, вы можете пометить свойство модификатором `lateinit`:

```
public class MyTest {
    lateinit var subject: TestSubject


    @SetUp fun setup() {
        subject = TestSubject()
    }


    @Test fun test() {
        subject.method()  // объект инициализирован, проверять на null не
нужно
```

```
    }
}
```

Такой модификатор может быть использован только с `var` свойствами, объявленными внутри тела класса (не в главном конструкторе). И только тогда, когда свойство не имеет пользовательских геттеров и сеттеров. Тип такого свойства должен быть non-null и не должен быть примитивным.

Доступ к `lateinit` свойству до того, как оно проинициализировано, выбрасывает специальное исключение, которое чётко обозначает, что свойство не было определено.

Переопределение свойств

См. Переопределение членов класса

Делегированные свойства

Самый распространённый тип свойств просто считывает (или записывает) данные из *backing field*. Тем не менее, с пользовательскими геттерами и сеттерами мы можем реализовать совершенно любое поведение свойства. В реальности, существуют общепринятые шаблоны того, как могут работать свойства. Несколько примеров:

- Вычисление значения свойства при первом доступе к нему (ленивые свойства)
- Чтение из ассоциативного списка с помощью заданного ключа
- Доступ к базе данных
- Оповещение listener'а в момент доступа и т.п.

Такие распространённые поведения свойств могут быть реализованы в виде библиотек с помощью *делегированных свойств*.

Интерфейсы

Интерфейсы в **Kotlin** очень похожи на интерфейсы в **Java 8**. Они могут содержать абстрактные методы, методы с реализацией. Главное отличие интерфейсов от абстрактных классов заключается в невозможности хранения переменных экземпляров. Они могут иметь свойства, но те должны быть либо абстрактными, либо предоставлять реализацию методов доступа.

Интерфейс определяется ключевым словом **interface**:

```
interface MyInterface {
    fun bar()
    fun foo() {
      // необязательное тело
    }
```

```
}
```

Реализация интерфейсов

Класс или объект могут реализовать любое количество интерфейсов:

```
class Child : MyInterface {
    override fun bar() {
        // тело
    }
}
```

Свойства в интерфейсах

Вы можете объявлять свойства в интерфейсах. Свойство, объявленное в интерфейсе, может быть либо абстрактным, либо иметь свою реализацию методов доступа. Свойства в интерфейсах не могут иметь *backing fields*, соответственно, методы доступа к таким свойствам не могут обращаться к *backing fields*.

```
interface MyInterface {
    val prop: Int // абстрактное свойство

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(prop)
    }
}

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

Устранение противоречий при переопределении

Когда мы объявлем большое количество типов в списке нашего супертипа, может так выйти, что мы допустим более одной реализации одного и того же метода. Например:

```
interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }
}
```

Оба интерфейса *A* и *B* объявляют функции *foo()* и *bar()*. Оба реализуют *foo()*, но только *B* содержит реализацию *bar()* (*bar()* не отмечен как абстрактный метод в интерфейсе *A*, потому что в интерфейсах это подразумевается по умолчанию, если у функции нет тела). Теперь, если мы унаследуем какой-нибудь класс *C* от *A*, нам, очевидно, придётся переопределять *bar()*, обеспечивать его реализацию. А если мы унаследуем *D* от *A* и *B*, нам не надо будет переопределять *bar()*, потому что мы унаследовали только одну его имплементацию. Но мы получили в наследство две имплементации *foo()*, поэтому компилятору не известно, какую выбрать. Он заставит нас переопределить функцию *foo()* и явно указать что мы имели ввиду.

Модификаторы доступа

Классы, объекты, интерфейсы, конструкторы, функции, свойства и их сеттеры могут иметь *модификаторы доступа* (у геттеров всегда такая же видимость, как у свойств, к

которым они относятся). В **Kotlin**предусмотрено четыре модификатора доступа: `private`, `protected`, `internal` и `public`. Если явно не используется никакого модификатора доступа, то по умолчанию применяется `public`.

Ниже вы найдёте описание всех возможных способов задавать область видимости.

Пакеты

Функции, свойства, классы, объекты и интерфейсы могут быть объявлены на самом "высоком уровне" прямо внутри пакета:

```
// имя файла: example.kt

package foo


fun baz() {}

class Bar {}
```

- Если вы не укажете никакого модификатора доступа, будет использован `public`. Это значит, что весь код данного объявления будет виден из космоса;
- Если вы пометите объявление словом `private`, оно будет иметь видимость только внутри файла, где было объявлено;
- Если вы используете `internal`, видимость будет распространяться на весь модуль;
- `protected` запрещено использовать в объявлениях "высокого уровня".

Примеры:

```
// file name: example.kt

package foo


private fun foo() {} // имеет видимость внутри example.kt


public var bar: Int = 5 // свойство видно со дна Марианской впадины

    private set         // сеттер видно только внутри example.kt


internal val baz = 6    // имеет видимость внутри модуля
```

Классы и интерфейсы

Для членов, объявленых в классе:

- `private` означает видимость только внутри этого класса (включая его членов);

- `protected` --- то же самое, что и `private` + видимость в субклассах;
- `internal` --- любой клиент *внутри модуля*, который видит объявленный класс, видит и его `internal` члены;
- `public` --- любой клиент, который видит объявленный класс, видит его `public` члены.

Примечание для Java программистов: в **Kotlin** внешний класс не видит `private`члены своих вложенных классов.
Если вы переопределите `protected` член и явно не укажете его видимость, переопределённый элемент также будет иметь модификатор доступа `protected`.

Примеры:

```
open class Outer {
    private val a = 1
    protected open val b = 2
    internal val c = 3
    val d = 4  // public по умолчанию

    protected class Nested {
        public val e: Int = 5
    }
}

class Subclass : Outer() {
    // a не видно
    // b, c и d видно
    // класс Nested и e видно

    override val b = 5   // 'b' - protected
}

class Unrelated(o: Outer) {
    // o.a, o.b не видно
    // o.c и o.d видно (тот же модуль)
    // Outer.Nested не видно, и Nested::e также не видно
```

```
}
```

Конструкторы

Для указания видимости главного конструктора класса используется следующий синтаксис (кстати, надо добавить ключевое слово *constructor*):

```
class C private constructor(a: Int) { ... }
```

В этом примере конструктор является `private`. По умолчанию все конструкторы имеют модификатор доступа `public`, то есть видны везде, где виден сам класс (а вот конструктор `internal` класса видно только в том же модуле).

Локальные объявления

Локальные переменные, функции и классы не могут иметь модификаторов доступа.

Модули

Модификатор доступа `internal` означает, что этот член видно в рамках его модуля. Модуль - это набор скомпилированных вместе **Kotlin** файлов:

- модуль в IntelliJ IDEA;
- Maven или Gradle проект;
- набор скомпилированных вместе файлов с одним способом вызова `<kotlinc>` задачи в Ant.

Расширения (extensions)

Аналогично таким языкам программирования, как **C#** и **Gosu**, **Kotlin** позволяет расширять класс путём добавления нового функционала. Не наследуясь от такого класса и не используя паттерн "Декоратор". Это реализовано с помощью специальных выражений, называемых *расширения*. **Kotlin** поддерживает *функции-расширения* и *свойства-расширения*.

Функции-расширения

Для того, чтобы объявить функцию-расширение, нам нужно указать в качестве приставки *возвращаемый тип*, то есть тип, который мы расширяем. Следующий пример добавляет функцию `swap` к `MutableList<Int>`:

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' даёт ссылку на Int
    this[index1] = this[index2]
    this[index2] = tmp
```

```
}
```

Ключевое слово *this* внутри функции-расширения соотносится с получаемым объектом (его тип ставится перед точкой). Теперь мы можем вызывать такую функцию в любом `MutableList<Int>`:

```
val l = mutableListOf(1, 2, 3)

l.swap(0, 2) // 'this' внутри 'swap()' не будет содержать значение 'l'
```

Разумеется, эта функция имеет смысл для любого `MutableList<T>`, и мы можем сделать её обобщённой:

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' относится к листу
    this[index1] = this[index2]
    this[index2] = tmp
}
```

Мы объявляем обобщённый тип-параметр перед именем функции для того, чтобы он был доступен в получаемом типе-выражении. См. Обобщения.

Расширения вычисляются статически

Расширения на самом деле не проводят никаких модификаций с классами, которые они расширяют. Объявляя расширение, вы создаёте новую функцию, а не новый член класса. Такие функции могут быть вызваны через точку, применимо к конкретному типу.

Мы хотели бы подчеркнуть, что расширения имеют статическую реализацию: это значит, что вызванная функция-расширение определена типом вызывающего её выражения, а не типом результата, который получается в ходе выполнения программы. К примеру:

```
open class C


class D: C()


fun C.foo() = "c"


fun D.foo() = "d"


fun printFoo(c: C) {
    println(c.foo())
}
```

```
printFoo(D())
```

Этот пример выведет нам "c" на экран потому, что вызванная функция-расширение зависит только от объявленного параметризованного типа `c`, который является `C` классом.

Если в классе есть и член в виде обычной функции, и функция-расширение с тем же возвращаемым типом, таким же именем и применяется с такими же аргументами, то **обычная функция в приоритете**. К примеру:

```
class C {
    fun foo() { println("member") }
}

fun C.foo() { println("extension") }
```

Если мы вызовем `c.foo()` любого объекта `c` с типом `C`, на экран выведется "member", а не "extension".

Однако, для экстеншн-функций совершенно нормально перегружать функции-члены, которые имеют такое же имя, но другую сигнатуру:

```
class C {
    fun foo() { println("member") }
}

fun C.foo(i: Int) { println("extension") }
```

Обращение к `C().foo(1)` выведет на экран надпись "extension".

Возвращаемое null значение

Обратите внимание, что расширения могут быть объявлены с возможностью получения null в качестве возвращаемого значения. Такие расширения могут ссылаться на переменные объекта, даже если их значение null. В таком случае есть возможность провести проверку `this == null` внутри тела функции. Благодаря этому метод `toString()` в языке **Koltin** вызывается без проверки на null: она проходит внутри функции-расширения.

```
fun Any?.toString(): String {
    if (this == null) return "null"
    // после проверки на null, `this` автоматически кастуется к не-null ти
пу, поэтому toString()
    // обращается (ориг.: resloves) к функции-члену класса Any
```

```
    return toString()
}
```

Свойства-расширения

Аналогично функциям, **Kotlin** поддерживает расширения свойств:

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

Так как расширения на самом деле не добавляют никаких членов к классам, свойство-расширение не может иметь *backing field*. Вот почему **запрещено использовать инициализаторы для свойств-расширений**. Их поведение может быть определено только явным образом, с указанием геттеров/сеттеров.

Пример:

```
val Foo.bar = 1 // ошибка: запрещено инициализировать значения в свойствах -расширениях
```

Расширения вспомогательных объектов (ориг.: *companion object extensions*)

Если у класса есть вспомогательный объект, вы также можете определить функции и свойства для такого объекта:

```
class MyClass {
    companion object { }  // называется "companion"
}


fun MyClass.Companion.foo() {
    // ...
}
```

Как и обычные члены вспомогательного объекта, они могут быть вызваны с помощью имени класса в качестве точки доступа:

```
MyClass.foo()
```

Область видимости расширений

Чаще всего мы объявляем расширения на самом верхнем уровне, то есть сразу под пакетами:

```
package foo.bar
```

```
fun Baz.goo() { ... }
```

Для того, чтобы использовать такое расширение вне пакета, в котором оно было объявлено, нам надо импортировать его на стороне вызова:

```
package com.example.usage


import foo.bar.goo // импортировать все расширения за именем "goo"
                   // или
import foo.bar.*   // импортировать все из "foo.bar"


fun usage(baz: Baz) {
    baz.goo()
)
```

См. Импорт для более подробной информации.

Объявление расширений в качестве членов класса

Внутри класса вы можете объявить расширение для другого класса. Внутри такого объявления существует несколько *неявных приёмников* (ориг.:*implicit receivers*), доступ к членам которых может быть произведён без классификатора. Экземпляр такого класса, к которому относится вызываемое расширение, называется *отсылкой* (ориг.: *dispatch receiver*), а экземпляр класса, в котором вызывается расширение называется *принимающим расширением* (ориг.: *extension receiver*).

```
class D {
    fun bar() { ... }
}


class C {
    fun baz() { ... }


    fun D.foo() {
        bar()   // вызывает D.bar
```

```
        baz()   // вызывает C.baz
    }

    fun caller(d: D) {
        d.foo()   // вызов функции-расширения
    }
}
```

В случае конфликта имён между членами класса, к которому отсылается расширение, и членами класса, в котором оно вызывается, в приоритете будут именна класса, принимающего расширение.

```
class C {
    fun D.foo() {
        toString()         // вызывает D.toString()
        this@C.toString()  // вызывает C.toString()
    }
```

Расширения, объявленные как члены класса, могут иметь модификатор видимости `open` и быть переопределены в унаследованных классах. Это означает, что виртуально такая отсылка происходит с учётом типа, к которому она отсылает, но статически - с учётом типа, возвращаемого таким расширением.

```
open class D {
}


class D1 : D() {
}


open class C {
    open fun D.foo() {
        println("D.foo in C")
    }

    open fun D1.foo() {
        println("D1.foo in C")
    }
```

```
    fun caller(d: D) {
        d.foo()   // вызов функции-расширения
    }
}

class C1 : C() {
    override fun D.foo() {
        println("D.foo in C1")
    }

    override fun D1.foo() {
        println("D1.foo in C1")
    }
}

C().caller(D())   // prints "D.foo in C"
C1().caller(D())  // prints "D.foo in C1" - получатель отсылки вычислен виртуально
C().caller(D1())  // prints "D.foo in C" - получатель расширения вычислен статически
```

Мотивация

В **Java** мы привыкли к классам с названием "*Utils": `FileUtils`, `StringUtils` и т.п. Довольно известным следствием этого является `java.util.Collections`. Но вот использование таких утилитных классов в своём коде - не самое приятное мероприятие:

```
// Java
Collections.swap(list, Collections.binarySearch(list, Collections.max(otherList)), Collections.max(list))
```

Имена таких классов постоянно используются при вызове. Мы можем их статически импортировать и получить что-то типа:

```
// Java
swap(list, binarySearch(list, max(otherList)), max(list))
```

Уже лучше, но такой мощный инструмент IDE, как автодополнение, не предоставляет нам сколь-нибудь серьёзную помощь в данном случае. Намного лучше, если бы у нас было:

```
// Java

list.swap(list.binarySearch(otherList.max()), list.max())
```

Но мы же не хотим реализовывать все методы класса `List`, так? Вот для чего и нужны расширения.

Классы данных

Нередко мы создаём классы, единственным назначением которых является хранение данных. Функционал таких классов зависит от самих данных, которые в них хранятся.
В **Kotlin** класс может быть отмечен словом `data`:

```
data class User(val name: String, val age: Int)
```

Такой класс называется *классом данных*. Компилятор автоматически извлекает все члены данного класса из свойств, объявленных в первичном конструкторе:

- пара функций `equals()`/`hashCode()`,
- `toString()` в форме `"User(name=Jhon, age=42)"`,
- функции componentN(), которые соответствуют свойствам, в зависимости от их порядка либо объявления,
- функция `copy()` (см. ниже)

Если какая-либо из этих функций явно определена в теле класса (или унаследована от родительского класса), то генерироваться она не будет.

Для того, чтобы поведение генерируемого кода соответствовало здравому смыслу, классы данных должны быть оформлены с соблюдением некоторых требований:

- Первичный конструктор должен иметь как минимум один параметр;
- Все параметры первичного конструктора должны быть отмечены, как `val` или `var`;
- Классы данных не могут быть абстрактными, open, sealed или inner;
- Дата-классы не могут наследоваться от других классов (но могут реализовывать интерфейсы).

Начиная с версии 1.1, классы данных могут расширять другие классы (см. примеры в Sealed classes)

Для того, чтобы у сгенерированного в JVM класса был конструктор без параметров, значения всех свойств должны быть заданы по умолчанию (см. Конструкторы)

```
 data class User(val name: String = "", val age: Int = 0)
 ```
```
```

Копирование

Довольно часто нам приходится копировать объект с изменением только *некоторых* его свойств. Для этой задачи генерируется функция `copy()`. Для написанного выше класса `User` такая реализация будет выглядеть следующим образом:

```
fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
```

Это позволяет нам писать

```
val jack = User(name = "Jack", age = 1)

val olderJack = jack.copy(age = 2)
```

Классы данных и мульти-декларации

Сгенерированные для классов данных *составные функции* позволяют использовать их в мульти-декларациях:

```
val jane = User("Jane", 35)

val (name, age) = jane

println("$name, $age years of age") // выводит "Jane, 35 years of age"
```

Стандартные классы данных

Стандартная библиотека предоставляет `Pair` и `Triple`. Однако, в большинстве случаев, проименованные классы данных являются лучшим решением, потому что делают код более читаемым, избегая малосодержательные имена для свойств.

Статья на эту тему на Хабре

Изолированные классы

Изолированные классы используются для отражения ограниченных иерархий классов, когда значение может иметь тип только из ограниченного набора, и никакой другой. Они являются, по сути, расширением enum-классов: набор значений enum типа также ограничен, но каждая enum-константа существует только в единственном экземпляре, в то время как наследник изолированного класса может иметь множество экземпляров, которые могут нести в себе какое-то состояние.

Чтобы описать изолированный класс, укажите модификатор `sealed` перед именем класса. Изолированный класс может иметь наследников, но все они должны быть объявлены в том же файле, что и сам изолированный класс. (До версии Kotlin 1.1 правила были ещё более строгими: классы должны были быть вложены в объявлении изолированного класса).

```
sealed class Expr
```

```
data class Const(val number: Double) : Expr()

data class Sum(val e1: Expr, val e2: Expr) : Expr()

object NotANumber : Expr()


fun eval(expr: Expr): Double = when (expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
}
```

(Пример выше использует одну новую возможность Kotlin 1.1: расширение классов, включая изолированные, классами данных) Обратите внимание, что классы, которые расширяют наследников изолированного класса (непрямые наследники) могут быть помещены где угодно, не обязательно в том же файле.

Ключевое преимущество от использования изолированных классов проявляется тогда, когда вы используете их в выражении when. Если возможно проверить что выражение покрывает все случаи, то вам не нужно добавлять `else`.

```
fun eval(expr: Expr): Double = when(expr) {
    is Expr.Const -> expr.number
    is Expr.Sum -> eval(expr.e1) + eval(expr.e2)
    Expr.NotANumber -> Double.NaN
    // оператор `else` не требуется, потому что мы покрыли все возможные с
лучаи
}
```

Обобщения (Generics)

Как и в **Java**, в **Kotlin** классы тоже могут иметь generic типы:

```
class Box<T>(t: T) {
    var value = t
}
```

Для того, чтобы создать объект такого класса, необходимо предоставить тип в качестве аргумента:

```
val box: Box<Int> = Box<Int>(1)
```

Но если параметры могут выведены из контекста (в аргументах конструктора или в некоторых других случаях), можно опустить указание типа:

```
val box = Box(1) // 1 имеет тип Int, поэтому компилятор отмечает для себя, что у переменной box тип — Box<Int>
```

Вариативность

Одним из самых сложных мест в системе типов **Java** являются маски (ориг. wildcards) (см. Java Generics FAQ). А в **Kotlin** этого нет. Вместо этого, у нас есть две другие вещи: *вариативность на уровне объявления* и *проекции типов*.

Для начала давайте подумаем на тему, зачем **Java** нужны эти странные маски. Проблема описана в книге Effective Java, Item 28: *Use bounded wildcards to increase API flexibility*. Обобщающие типы в **Java**, прежде всего, **неизменны**. Это значит, что `List<String>` *не является* подтипом `List<Object>`. Почему так? Если бы List был изменяемым, единственно лучшим решением для следующей задачи был бы массив, потому что после компиляции данный код вызвал бы ошибку в рантайме:

```
// Java
List<String> strs = new ArrayList<String>();
List<Object> objs = strs; // !!! Причина вышеуказанной проблемы заключена здесь, Java запрещает так делать
objs.add(1); // Тут мы помещаем Integer в список String'ов
String s = strs.get(0); // !!! ClassCastException: не можем кастовать Integer к String
```

Таким образом, **Java** запрещает подобные вещи, гарантируя тем самым безопасность в период выполнения кода. Но у такого подхода есть свои последствия. Рассмотрим, например, метод `addAll`интерфейса `Collection`. Какова сигнатура данного метода? Интуитивно мы бы указали её таким образом:

```
// Java
interface Collection<E> ... {
  void addAll(Collection<E> items);
}
```

Но тогда мы бы не могли выполнять следующую простую операция (которая является абсолютно безопасной):

```
// Java
void copyAll(Collection<Object> to, Collection<String> from) {
  to.addAll(from); // !!! Не скомпилируется с нативным объявлением метода addAll:
                   //       Collection<String> не является подтипом Collection<Object>
}
```

(В **Java** нам этот урок дорого стоил, см. Effective Java, Item 25: *Prefer lists to arrays*)

Вот почему сигнатура `addAll()` на самом деле такая:

```
// Java
interface Collection<E> ... {
  void addAll(Collection<? extends E> items);
}
```

Маска для аргумента `? extends T` указывает на то, что это метод принимает коллекцию объектов *некого типа* `T`, а не сам `T`. Это значит, что мы можем безопасно **читать** объекты типа `T` из содержимого (элементы коллекции являются экземплярами подкласса T), но **не можем их изменять**, потому что не знаем, какие объекты соответствуют этому неизвестному типу `T`. Минуя это ограничение, мы достигаем желаемого результата: `Collection<String>` **является** подтипом `Collection<? extends Object>`. Выражаясь более "умными словами", маска с **extends**-связкой (**верхнее** связывание) делает тип ковариантным (ориг. covariant).

Ключом к пониманию, почему этот трюк работает, является довольно простая мысль: использование коллекции `String`'ов и чтение из неё `Object`ов нормально только в случае, если вы **берёте** элементы из коллекции. Наоборот, если вы только *вносите* элементы в коллекцию, то нормально брать коллекцию `Object`'ов и помещать в неё `String`и: в **Java** есть `List<? super String>`, **супертип**`List<Object>`'а.

Это называется **контрвариантностью**. В `List<? super String>` вы можете вызвать только те методы, которые принимают String в качестве аргумента (например, `add(String)` или `set(int, String)`). В случае, если вы вызываете из `List<T>` что-то с возвращаемым значением `T`, вы получаете не `String`, а `Object`.

Джошуа Блок (Joshua Block) называет объекты:

- **Производителями** (ориг.:*producers*), если вы только **читаете** из них
- **Потребителями** (ориг.: *consumers*), если вы только **записываете** в них Его рекомендация: "*Для максимальной гибкости используйте маски (ориг. wildcards) на входных параметрах, которые представляют производителей или потребителей*"

PECS настаивает на Producer-Extends, Consumer-Super.

>*Примечание*: если вы используете объект-производитель, предположим, `List<? extends Foo>`, вы не можете вызвать методы `add()` или `set()` этого объекта. Но это не значит, что объект является **неизменяемым** (immutable): ничто не мешает вам вызвать метод `clear()` для того, чтобы очистить список, так как `clear()` не имеет аргументов. Единственное, что гарантируют маски — **безопасность типов**. Неизменяемость (ориг.: *immutability*) — совершенно другая история.

Вариантность на уровне объявления

Допустим, у нас есть generic интерфейс `Source<T>`, у которого нет методов, которые принимают `T` в качестве аргумента. Только методы, возвращающие `T`:

```
// Java
```

```
interface Source<T> {

  T nextT();

}
```

Тогда было бы вполне безопасно хранить ссылки на экземляр `Source<String>` в переменной типа `Source<Object>` — не нужно вызывать никакие методы-потребители. Но Java не знает этого и не воспринимает такой код:

```
// Java

void demo(Source<String> strs) {

  Source<Object> objects = strs; // !!! Запрещено в Java

  // ...

}
```

Чтобы исправить это, нам нужно объявить объекты типа `Source<? extends Object>`, что в каком-то роде бессмысленно, потому что мы можем вызывать у переменных только те методы, что и ранее, стало быть более сложный тип не добавляет значения. Но компилятор не знает этого.

В Kotlin существует способ объяснить вещь такого рода компилятору. Он называется **вариантность на уровне объявления**: мы можем пометить аннотацией **параметризованный тип** `T` класса `Source`, чтобы удостовериться, что он только **возвращается** (производится) членами `Source<T>`, и никогда не потребляется. Чтобы сделать это, нам необходимо использовать модификатор **out**

```
abstract class Source<out T> {

    abstract fun nextT(): T

}


fun demo(strs: Source<String>) {

    val objects: Source<Any> = strs // Всё в порядке, т.к. T — out-парамет
p

    // ...

}
```

Общее правило таково: когда параметр `T` класса `C` объявлен как **out**, он может использоваться только в **out**-местах в членах `C`. Но зато `C<Base>` может быть родителем `C<Derived>`, и это будет безопасно.

Говоря "умными словами", класс `C` **ковариантен** в параметре `T`;
или: `T` является **ковариантным**параметризованным типом.

Модификатор **out** называют **вариативной аннотацией**, и так как он указывается на месте объявления типа параметра, речь идёт о **вариативности на месте объявления**.

Эта концепция противопоставлена **вариативности на месте использования** из Java, где маски при использовании типа делают типы ковариантными.

В дополнении к **out**, Kotlin предоставляет дополнительную вариативную аннотацию **in**. Она делает параметризованный тип **контравариантным**: он может только потребляться, но не может производиться. `Comparable` является хорошим примером такого класса:

```
abstract class Comparable<in T> {
    abstract fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0 имеет тип Double, расширяющий Number
    // Таким образом, мы можем присвоить значение x переменной типа Comparable<Double>
    val y: Comparable<Double> = x // OK!
}
```

Мы верим, что слова **in** и **out** говорят сами за себя (так как они довольно успешно используются в C# уже долгое время), таким образом, мнемоника, приведённая выше, не так уж и нужна, и её можно перефразировать следущим образом:

Экзистенциональная Трансформация: Consumer in, Producer out! :-)

Проекции типов

Вариативность на месте использования

Объявлять параметризованный тип `T` как **out** очень удобно: при его использовании не будет никаких проблем с подтипами. И это действительно так в случае с классами, которые могут быть ограничены на только возвращение `T`. А как быть с теми классами, которые ещё и принимают `T`? Пример: класс `Array`

```
class Array<T>(val size: Int) {
    fun get(index: Int): T { /* ... */ }
    fun set(index: Int, value: T) { /* ... */ }
}
```

Этот класс не может быть ни ко-, ни контравариантным в `T`, что ведёт к некоторому снижению гибкости. Рассмотрим следующую функцию:

```
fun copy(from: Array<Any>, to: Array<Any>) {
    assert(from.size == to.size)
```

```
    for (i in from.indices)
        to[i] = from[i]
}
```

По задумке, это функция должна копировать значения из одного массива в другой. Давате попробуем сделать это на практике:

```
val ints: Array<Int> = arrayOf(1, 2, 3)
val any = Array<Any>(3)
copy(ints, any) // Ошибка: ожидалось (Array<Any>, Array<Any>)
```

Здесь мы попадаем в уже знакомую нам проблему: `Array<T>` **инвариантен** в `T`, таким образом `Array<Int>` не является подтипом `Array<Any>`. Почему? Опять же, потому что копирование **может**сотворить плохие вещи, например может произойти попытка **записать**, скажем, значение типа `String` в `from`. И если мы на самом деле передадим туда массив `Int`, через некоторое время будет выборошен `ClassCastException`.

Тогда единственная вещь, в которой мы хотим удостовериться, это то, что `copy()` не сделает ничего плохого. Мы хотим запретить методу **записывать** в `from`, и мы можем это сделать:

```
fun copy(from: Array<out Any>, to: Array<Any>) {
 // ...
}
```

Произошедшее здесь называется **проекция типов**: мы сказали, что `from` — не просто массив, а ограниченный (**спроецированный**): мы можем вызывать только те методы, которые возвращают параметризованный тип `T`, что в этом случае означает, что мы можем вызывать только `get()`. Таков наш подход к **вариативности на месте использования**, и он соответствует `Array<? extends Object>` из Java, но в более простом виде.

Вы так же можете проецировать тип с **in**:

```
fun fill(dest: Array<in String>, value: String) {
    // ...
}
```

`Array<in String>` соответствует `Array<? super String>` из Java, то есть мы можем передать массив `CharSequence` или массив `Object` в функцию `fill()`.

"Звёздные" проекции

Иногда возникает ситуация, когда вы ничего не знаете о типе аргумента, но всё равно хотите использовать его безопасным образом. Этой безопасности можно добиться путём определения такой проекции параметризованного типа, при которой его экземпляр будет подтипом этой проекции.

Kotlin предоставляет так называемый **star-projection** синтаксис для этого:

- Для `Foo<out T>`, где `T` — ковариантный параметризованный тип с верхней границей `TUpper`, `Foo<*>` является эквивалентом `Foo<out TUpper>`. Это значит, что когда `T` неизвестен, вы можете безопасно *читать* значения типа `TUpper` из `Foo<*>`.
- Для `Foo<in T>`, где `T` — ковариантный параметризованный тип, `Foo<*>` является эквивалентом `Foo<in Nothing>`. Это значит, что вы не можете безопасно *писать* в `Foo<*>` при неизвестном `T`.
- Для `Foo<T>`, где `T` — инвариантный параметризованный тип с верхней границей `TUpper`, `Foo<*>` является эквивалентом `Foo<out TUpper>` при чтении значений и `Foo<in Nothing>` при записи значений.

Если параметризованный тип имеет несколько параметров, каждый из них проецируется независимо. Например, если тип объявлен как `interface Function<in T, out U>`, мы можем представить следующую "звёздную" проекцию:

- `Function<*, String>` означает `Function<in Nothing, String>`;
- `Function<Int, *>` означает `Function<Int, out Any?>`;
- `Function<*, *>` означает `Function<in Nothing, out Any?>`.

  Примечание: "звёздные" проекции очень похожи на сырые (raw) типы из Java, за тем исключением, что являются безопасными.

Обобщённые функции

Функции, как и классы, могут иметь типовые параметры. Типовые параметры помещаются перед именем функции:

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


fun <T> T.basicToString() : String {  // функция-расширение
    // ...
}
```

Для вызова обобщённой функции, укажите тип аргументов на месте вызова **после** имени функции:

```
val l = singletonList<Int>(1)
```

Обобщённые ограничения

Набор всех возможных типов, которые могут быть переданы в качестве параметра, может быть ограничен с помощью **обобщённых ограничений**.

Верхние границы

Самый распространённый тип ограничений - **верхняя граница**, которая соответствует ключевому слову *extends* из Java:

```
fun <T : Comparable<T>> sort(list: List<T>) {
    // ...
}
```

Тип, указанный после двоеточия, является **верхней границей**: только подтип `Comparable<T>` может быть передан в `T`. Например:

```
sort(listOf(1, 2, 3)) // Всё в порядке. Int — подтип Comparable<Int>

sort(listOf(HashMap<Int, String>())) // Ошибка: HashMap<Int, String> не является подтипом Comparable<HashMap<Int, String>>
```

По умолчанию (если не указана явно) верняя граница — `Any?`. Только одна верхняя граница может быть указана в угловых скобках. В случае, если один параметризованный тип требует больше чем одной верхней границы, нам нужно использовать разделяющее **where**-условие:

```
fun <T> cloneWhenGreater(list: List<T>, threshold: T): List<T>
    where T : Comparable,
          T : Cloneable {
  return list.filter { it > threshold }.map { it.clone() }
}
```

Вложенные классы

Классы могут быть вложены в другие классы

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}


val demo = Outer.Nested().foo() // == 2
```

Внутренние классы

Класс может быть отмечен как внутренний с помощью слова `inner`, тем самым он будет иметь доступ к членам внешнего класса. Внутренние классы содержат ссылку на объект внешнего класса:

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

Подробнее об использовании `this` во внутренних классах: Qualified `this` expressions

Анонимные внутренние классы

Анонимные внутренние экземпляры классов создаются с помощью object expression:

```
window.addMouseListener(object: MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // ...
    }

    override fun mouseEntered(e: MouseEvent) {
        // ...
    }
})
```

Если объект является экземпляром функционального Java-интерфейса (т.е. Java-интерфейса с единственным абстрактным методом), вы можете создать его с помощью лямбда-выражения с префиксом — типом интерфейса:

```
val listener = ActionListener { println("clicked") }
```

Перечисляемые типы

Наиболее базовый пример использования enum — это реализация типобезопасных перечислений

```
enum class Direction {
```

```
    NORTH, SOUTH, WEST, EAST
}
```

Каждая enum-константа является объектом. При объявлении константы разделяются запятыми.

Инициализация

Так как константы являются экземплярами enum-класса, они могут быть инициализированы

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

Анонимные классы

Enum-константы также могут объявлять свои собственные анонимные классы

```
enum class ProtocolState {
    WAITING {
        override fun signal() = TALKING
    },

    TALKING {
        override fun signal() = WAITING
    };

    abstract fun signal(): ProtocolState
}
```

как с их собственными методами, так и с перегруженными методами базового класса. Следует заметить, что при объявлении в enum-классе каких-либо членов, необходимо отделять их от списка констант точкой с запятой, так же как и в Java.

Работа с enum-константами

Так же как и в Java, enum-классы в Kotlin имеют стандартные методы для вывода списка объявленных констант и для получения enum-константы по её имени. Ниже приведены сигнатуры этих методов:

```
EnumClass.valueOf(value: String): EnumClass

EnumClass.values(): Array<EnumClass>
```

Метод `valueOf()` выбрасывает исключение `IllegalArgumentException`, если указанное имя не соответствует ни одной константе, объявленной в классе.

Каждая enum-константа имеет поля, в которых содержатся её имя и порядковый номер в enum-классе:

```
val name: String

val ordinal: Int
```

Также enum-константы реализуют интерфейс Comparable. Порядок сортировки соответствует порядку объявления.

Анонимные объекты и объявление объектов

Иногда нам необходимо получить экземпляр некоторого класса с незначительной модификацией, желательно без написания нового подкласса. **Java** справляется с этим с помощью *вложенных анонимных классов*. **Kotlin** несколько улучшает данный подход.

Анонимные объекты (ориг.:*Object expressions*)

Для того, чтобы создать объект анонимного класса, который наследуется от какого-то типа (типов), используется конструкция:

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // ...
    }

    override fun mouseEntered(e: MouseEvent) {
        // ...
    }
})
```

Если у супертипа есть конструктор, то в него должны быть переданы соответсвующие параметры. Множество супертипов может быть указано после двоеточия в виде списка, заполненного через запятую:

```
open class A(x: Int) {
    public open val y: Int = x
}

interface B {...}

val ab: A = object : A(1), B {
    override val y = 15
}
```

Если всё-таки нам нужен *просто объект* без всяких там родительских классов, то можем указать:

```
val adHoc = object {
    var x: Int = 0
    var y: Int = 0
}
print(adHoc.x + adHoc.y)
```

Код внутри объявленного объекта может обращаться к переменным за скобками так же, как вложенные анонимные классы в **Java**

```
fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0

    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }

        override fun mouseEntered(e: MouseEvent) {
            enterCount++
        }
```

```
    })
    // ...
}
```

Объявления объектов (ориг.:*Object declarations*)

Синглтон - очень полезный паттерн программирования, и **Kotlin** (переняв у **Scala**) позволяет объявлять его довольно простым способом :

```
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ...
    }

    val allDataProviders: Collection<DataProvider>
        get() = // ...
}
```

Это называется *объявлением объекта* и всегда имеет приставку в виде ключевого слова **object**. Аналогично объявлению переменной, объявление объекта не является выражением и не может быть использовано в правой части оператора присваивания.

Для непосредственной ссылки на объект используется его имя:

```
DataProviderManager.registerDataProvider(...)
```

Подобные объекты могут иметь супертипы:

```
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // ...
    }

    override fun mouseEntered(e: MouseEvent) {
        // ...
    }
}
```

ПРИМЕЧАНИЕ: объявление объекта не может иметь локальный характер (т.е. быть вложенным непосредственно в функцию), но может быть вложено в объявление другого объекта или какого-либо невложенного класса.

Вспомогательные объекты

Объявление объекта внутри класса может быть отмечено ключевым словом **companion**:

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

Для вызова членов такого `companion` объекта используется имя класса:

```
val instance = MyClass.create()
```

Не обязательно указывать имя вспомогательного объекта. В таком случае он будет назван `Companion`:

```
class MyClass {
    companion object {
    }
}


val x = MyClass.Companion
```

Такие члены вспомогательных объектов выглядят, как статические члены в других языках программирования. На самом же деле, они являются членами реальных объектов и могут реализовывать, к примеру, интерфейсы:

```
interface Factory<T> {
    fun create(): T
}



class MyClass {
    companion object : Factory<MyClass> {
        override fun create(): MyClass = MyClass()
    }
```

```
}
```

Однако в **JVM** вы можете статически генерировать методы вспомогательных объектов и полей, используя аннотацию `@JvmStatic@`. См. Совместимость с Java.

Семантическое различие между анонимным объектом и декларируемым объектом.

Существует только одно смысловое различие между этими двумя понятиями:

- анонимный объект инициализируется **сразу после того**, как был использован
- декларированный объект инициализируется **лениво**, в момент первого к нему доступа
- вспомогательный объект инициализируется в момент, когда класс, к которому он относится, загружен и семантически совпадает со статическим инициализатором **Java**

Делегирование

Делегирование класса

Шаблон делегирования является хорошей альтернативой наследованию, и Kotlin поддерживает его нативно, освобождая вас от необходимости написания шаблонного кода.

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main(args: Array<String>) {
    val b = BaseImpl(10)
    Derived(b).print() // prints 10
}
```

Ключевое слово `by` в оглавлении `Derived`, находящееся после типа делегируемого класса, говорит о том, что объект `b` типа `Base` будет храниться внутри экземпляра `Derived`, и компилятор сгенерирует у `Derived` соответствующие методы из `Base`, которые при вызове будут переданы объекту `b`

Делегированные свойства

За помощь в переводе спасибо официальному блогу JetBrains на Хабрахабре

Существует несколько основных видов свойств, которые мы реализовываем каждый раз вручную в случае их надобности. Однако намного удобнее было бы реализовать их раз и навсегда и положить в какую-нибудь библиотеку. Примеры таких свойств:

- ленивые свойства (lazy properties): значение вычисляется один раз, при первом обращении
- свойства, на события об изменении которых можно подписаться (observable properties)
- свойства, хранимые в ассоциативном списке, а не в отдельных полях

Для таких случаев, Kotlin поддерживает *делегированные свойства*:

```
class Example {

    var p: String by Delegate()

}
```

Их синтаксис выглядит следующим образом: `val/var <имя свойства>: <Тип> by <выражение>`. Выражение после *by* — *делегат*: обращения (`get()`, `set()`) к свойству будут обрабатываться этим выражением. Делегат не обязан реализовывать какой-то интерфейс, достаточно, чтобы у него были методы `get()` и `set()` с определённой сигнатурой:

```
class Delegate {

    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {

        return "$thisRef, спасибо за делегирование мне '${property.name}'!
"

    }



    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: St
ring) {

        println("$value было присвоено значению '${property.name} в $thisR
ef.'")

    }

}
```

Когда мы читаем значение свойства `p`, вызывается метод `getValue()` класса `Delegate`, причем первым параметром ей передается тот объект, у которого запрашивается свойство `p`, а вторым — объект-описание самого свойства p (у него можно, в частности, узнать имя свойства). Например:

```
val e = Example()

println(e.p)
```

Этот код выведет

```
Example@33a17727, спасибо за делегирование мне ‘p’!
```

Похожим образом, когда мы обращаемся к `p`, вызывается метод `setValue()`. Два первых параметра — такие же, как у get(), а третий — присваиваемое значение свойства:

```
e.p = "NEW"
```

Этот код выведет

```
NEW было присвоено значению ‘p’ в Example@33a17727.
```

Спецификация требований к делегированным свойствам может быть найдена ниже.

Заметьте, что начиная с версии Kotlin 1.1, вы можете объявлять делегированные свойства внутри функций или блоков кода, а не только внутри классов. Снизу вы можете найти пример.

Стандартные делегаты

Стандартная библиотека Kotlin предоставляет несколько полезных видов делегатов:

Ленивые свойства (lazy properties)

`lazy()` это функция, которая принимает лямбду и возвращает экземпляр класса `Lazy<T>`, который служит делегатом для реализации ленивого свойства: первый вызов `get()` запускает лямбда-выражение, переданное `lazy()` в качестве аргумента, и запоминает полученное значение, а последующие вызовы просто возвращают вычисленное значение.

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}


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

```
}
```

Этот код выведет:

```
computed!

Hello

Hello
```

По умолчанию вычисление ленивых свойств **синхронизировано**: значение вычисляется только в одном потоке выполнения, и все остальные потоки могут видеть одно и то же значение. Если синхронизация не требуется, передайте `LazyThreadSafetyMode.PUBLICATION` в качестве параметра в функцию `lazy()`, тогда несколько потоков смогут исполнять вычисление одновременно. Или если вы уверены, что инициализация всегда будет происходить в одном потоке исполнения, вы можете использовать режим `LazyThreadSafetyMode.NONE`, который не гарантирует никакой потокобезопасности.

Observable свойства

Функция `Delegates.observable()` принимает два аргумента: начальное значение свойства и обработчик (лямбда), который вызывается при изменении свойства. У обработчика три параметра: описание свойства, которое изменяется, старое значение и новое значение.

```
import kotlin.properties.Delegates


class User {
    var name: String by Delegates.observable("<no name>") {
        prop, old, new ->
        println("$old -> $new")
    }
}


fun main(args: Array<String>) {
    val user = User()
    user.name = "first"
    user.name = "second"
}
```

Этот код выведет:

```
<no name> -> first
```

```
first -> second
```

Если Вам нужно иметь возможность запретить присваивание некоторых значений, используйте функцию `vetoable()` вместо `observable()`.

Хранение свойств в ассоциативном списке

Один из самых частых сценариев использования делегированных свойств заключается в хранении свойств в ассоциативном списке. Это полезно в "динамическом" коде, например, при работе с JSON:

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}
```

В этом примере конструктор принимает ассоциативный список

```
val user = User(mapOf(
    "name" to "John Doe",
    "age"  to 25
))
```

Делегированные свойства берут значения из этого ассоциативного списка (по строковым ключам)

```
println(user.name) // Prints "John Doe"
println(user.age)  // Prints 25
```

Также, если вы используете `MutableMap` вместо `Map`, поддерживаются изменяемые свойства (var):

```
class MutableUser(val map: MutableMap<String, Any?>) {
    var name: String by map
    var age: Int     by map
}
```

Локальные делегированные свойства (с версии 1.1)

Вы можете объявить локальные переменные как делегированные свойства. Например, вы можете сделать локальную переменную ленивой:

```
fun example(computeFoo: () -> Foo) {
    val memoizedFoo by lazy(computeFoo)

    if (someCondition && memoizedFoo.isValid()) {
        memoizedFoo.doSomething()
    }
}
```

Переменная `memoizedFoo` будет вычислена только при первом обращении к ней. Если условие `someCondition` будет ложно, значение переменной не будет вычислено вовсе.

Требования к делегированным свойствам

Здесь приведены требования к объектам-делегатам.

Для **read-only** свойства (например *val*), делегат должен предоставлять функцию `getValue`, которая принимает следующие параметры:

- `thisRef` — должен иметь такой же тип или быть наследником типа *хозяина свойства* (для расширений — тип, который расширяется)
- `property` — должен быть типа `KProperty<*>` или его родительского типа. Эта функция должна возвращать значение того же типа, что и свойство (или его родительского типа).

Для **изменяемого** свойства (*var*) делегат должен *дополнительно* предоставлять функцию `setValue`, которая принимает следующие параметры:

- `thisRef` — то же что и у `getValue()`,
- `property` — то же что и у `getValue()`,
- new value — должен быть того же типа, что и свойство (или его родительского типа).

Функции `getValue()` и/или `setValue()` могут быть предоставлены либо как члены класса-делегата, либо как его расширения. Последнее полезно когда вам нужно делегировать свойство объекту, который изначально не имеет этих функций. Обе эти функции должны быть отмечены с помощью ключевого слова `operator`.

Эти интерфейсы объявлены в стандартной библиотеке Kotlin:

```
interface ReadOnlyProperty<in R, out T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
}


interface ReadWriteProperty<in R, T> {
```

```
    operator fun getValue(thisRef: R, property: KProperty<*>): T
    operator fun setValue(thisRef: R, property: KProperty<*>, value: T)
}
```

Translation Rules

Для каждого делегированного свойства компилятор Kotlin "за кулисами" генерирует вспомогательное свойство и делегирует его. Например, для свойства `prop` генерируется скрытое свойство `prop$delegate`, и исполнение геттеров и сеттеров просто делегируется этому дополнительному свойству:

```
class C {
    var prop: Type by MyDelegate()
}

// этот код генерируется компилятором:
class C {
    private val prop$delegate = MyDelegate()
    var prop: Type
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::prop, value)
}
```

Компилятор Kotlin предоставляет всю необходимую информацию о `prop` в аргументах: первый аргумент `this` ссылается на экземпляр внешнего класса `C` и `this::prop` reflection-объект типа `KProperty`, описывающий сам `prop`.

Заметьте, что синтаксис `this::prop` для обращения к bound callable reference напрямую в коде программы доступен только с Kotlin версии 1.1

Предоставление делегата

Примечание: Предоставление делегата доступно в Kotlin начиная с версии 1.1

С помощью определения оператора `provideDelegate` вы можете расширить логику создания объекта, которому будет делегировано свойство. Если объект, который используется справа от `by`, определяет `provideDelegate` как член или как расширение, эта функция будет вызвана для создания экземпляра делегата.

Один из возможных юзкейсов `provideDelegate` — это проверка состояния свойства при его создании.

Например, если вы хотите проверить имя свойства перед связыванием, вы можете написать что-то вроде:

```
class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(
            thisRef: MyUI,
            prop: KProperty<*>
    ): ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, prop.name)
        // создание делегата
    }

    private fun checkProperty(thisRef: MyUI, name: String) { ... }
}

fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { ... }

class MyUI {
    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}
```

`provideDelegate` имеет те же параметры, что и `getValue`:

- `thisRef` — должен иметь такой же тип, или быть наследником типа *хозяина свойства* (для расширений — тип, который расширяется)
- `property` — должен быть типа `KProperty<*>` или его родительского типа. Эта функция должна возвращать значение того же типа, что и свойство (или его родительского типа)

Метод `provideDelegate` вызывается для каждого свойства во время создания экземпляра `MyUI`, и сразу совершает необходимые проверки.

Не будь этой возможности внедрения между свойством и делегатом, для достижения той же функциональности вам бы пришлось передавать имя свойства явно, что не очень удобно:

```
// Проверяем имя свойства без "provideDelegate"
class MyUI {
    val image by bindResource(ResourceID.image_id, "image")
    val text by bindResource(ResourceID.text_id, "text")
```

```
}


fun <T> MyUI.bindResource(
        id: ResourceID<T>,
        propertyName: String
): ReadOnlyProperty<MyUI, T> {
   checkProperty(this, propertyName)
   // создание делегата
}
```

В сгенерированном коде метод `provideDelegate` вызывается для инициализации вспомогательного свойства `prop$delegate`. Сравните сгенерированный для объявления свойства код `val prop: Type by MyDelegate()` со сгенерированным кодом из Transaction Rules (когда `provideDelegate` не представлен):

```
class C {
    var prop: Type by MyDelegate()
}


// этот код будет сгенерирован компилятором
// когда функция 'provideDelegate' доступна:
class C {
    // вызываем "provideDelegate" для создания вспомогательного свойства "delegate"
    private val prop$delegate = MyDelegate().provideDelegate(this, this::prop)
    val prop: Type
        get() = prop$delegate.getValue(this, this::prop)
}
```

Заметьте, что метод `provideDelegate` влияет только на создание вспомогательного свойства и не влияет на код, генерируемый геттером или сеттером.

Функции

Объявление функций

В **Kotlin** функции объявляются с помощью ключевого слова *fun*

```
fun double(x: Int): Int {

}
```

Применение функций

При вызове функции используется традиционный подход

```
val result = double(2)
```

Для вызова вложенной функции используется знак точки

```
Sample().foo() //создаёт экземпляр класса Sample и вызывает foo
```

Инфиксная запись

Функции так же могут быть вызваны при помощи инфиксной записи, при условии, что:

- Они являются членом другой функции или расширения
- В них используется один параметр
- Когда они помечены ключевым словом `infix`

```
// Определяем выражение как Int
infix fun Int.shl(x: Int): Int {
...
}


// вызываем функцию, используя инфиксную запись


1 shl 2


// то же самое, что


1.shl(2)
```

Параметры

Параметры функции записываются аналогично системе обозначений в языке Pascal, *имя*:*тип*. Параметры разделены запятыми. Каждый параметр должен быть явно указан.

```
fun powerOf(number: Int, exponent: Int) {
```

```
...
}
```

Аргументы по умолчанию

Параметры функции могут иметь значения по умолчанию, которые используются в случае, если аргумент функции не указан при её вызове. Это позволяет снизить уровень перегруженности кода по сравнению с другими языками.

```
fun read(b: Array<Byte>, off: Int = 0, len: Int = b.size()) {
...
}
```

Значения по умолчанию указываются после типа знаком **=**.

Переопределённые методы всегда используют те же самые значения по умолчанию, что и их базовые методы. При переопределении методов со значениями по умолчанию эти параметры должны быть опущены:

```
open class A {
    open fun foo(i: Int = 10) { ... }
}

class B : A() {
    override fun foo(i: Int) { ... }  // значение по умолчанию указать нельзя
}
```

Имена в названиях аргументов

Параметры функции могут быть названы в момент вызова функций. Это очень удобно, когда у функции большой список параметров, в том числе со значениями по умолчанию.

Рассмотрим такую функцию

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = false,
             wordSeparator: Char = ' ') {
...
}
```

мы можем вызвать её, используя аргументы по умолчанию

```
reformat(str)
```

Однако, при вызове этой функции без аргументов по умолчанию, получится что-то вроде

```
reformat(str, true, true, false, '_')
```

С названными аргументами мы можем сделать код намного более читаемым

```
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
```

Или, если нам не нужны все эти аргументы

```
reformat(str, wordSeparator = '_')
```

Обратите внимание, что синтаксис названных аргументов не может быть использован при вызове **Java**функций, потому как байт-код **Java** не всегда хранит имена параметров функции.

Функции с возвращаемым типом `Unit`

Если функция не возвращает никакого полезного значения, её возвращаемый тип - `Unit`. *`Unit` - тип только с одним значением -* `Unit`. Это возвращаемое значение не нуждается в явном указании

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
    else
        println("Hi there!")
    // `return Unit` или `return` необязательны
}
```

Указание типа `Unit` в качестве возвращаемого значения тоже не является обязательным. Код, написанный выше, совершенно идентичен с

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

Функции с одним выражением

Когда функция возвращает одно-единственное выражение, фигурные скобки `{ }` могут быть опущены, и тело функции может быть описано после знака `=`

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

Компилятор способен сам определить типа возвращаемого значения.

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

Явные типы возвращаемых значений

Функции, в которых есть тело, всегда должны указывать возвращаемый ими тип данных (если в этом качестве не указан тип `Unit`). **Kotlin** не вычисляет самостоятельно тип возвращаемого значения для функций с заключённым в них блоком кода потому, что подобные функции могут иметь сложную структуру и возвращаемый тип неочевиден для читающего этот код человека (иногда даже для компилятора).

Нефиксированное число аргументов (Varargs)

Параметр функции (обычно для этого используется последний) может быть помечен модификатором `vararg`:

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts - это массив (Array)
        result.add(t)
    return result
}
```

это позволит указать множество значений в качестве аргументов функции:

```
val list = asList(1, 2, 3)
```

Внутри функции параметр с меткой `vararg` и типом `T` виден как массив элементов `T`, таким образом переменная `ts` в вышеуказанном примере имеет тип `Array<out T>`.

Только один параметр может быть помечен меткой `vararg`. Если параметр с именем `vararg` не стоит на последнем месте в списке аргументов, значения для соответствующих параметров могут быть переданы с использованием *named argument* синтаксиса. В случае, если параметр является функцией, для этих целей можно вынести лямбду за фигурные скобки.

При вызове `vararg` функции мы можем передать аргументы один-за-одним, например `asList(1, 2, 3)`, или, если у нас уже есть необходимый массив элементов и мы хотим передать его содержимое в нашу функцию, использовать оператор *spread* (необходимо пометить массив знаком `*`):

```
val a = arrayOf(1, 2, 3)

val list = asList(-1, 0, *a, 4)
```

Область действия функций

В **Kotlin** функции могут быть объявлены в самом начале файла. Подразумевается, что вам не обязательно создавать объект какого-либо класса, чтобы воспользоваться его функцией (как в **Java**, **C#** или **Scala**). В дополнение к этому, функции в языке **Kotlin** могут быть объявлены локально, как функции-члены (ориг. *"member functions"*) и функции-расширения (*"extension functions"*).

Локальные функции

Koltin поддерживает локальные функции. Например, функции, вложенные в другие функции

```
fun dfs(graph: Graph) {
    fun dfs(current: Vertex, visited: Set<Vertex>) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v, visited)
    }

    dfs(graph.vertices[0], HashSet())
}
```

Такие локальные функции могут иметь доступ к локальным переменным внешних по отношению к ним функций (типа *closure*). Таким образом, в примере, приведённом выше, *visited* может быть локальной переменной.

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0])
```

```
}
```

Функции-элементы

Функции-элементы - это функции, объявленные внутри классов или объектов

```
class Sample() {
    fun foo() { print("Foo") }
}
```

Функции-элементы вызываются с использованием точки

```
Sample().foo() // создаёт инстанс класса Sample и вызвает его функцию foo
```

Для более подробной информации о классах и их элементах см. Классы

Функции-обобщения (*Generic Functions*)

Функции могут иметь обобщённые параметры, которые задаются треугольными скобками и помещаются перед именем функции

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

Для более подробной информации об обобщениях см. Обобщения

Встроенные функции (*Inline Functions*)

О встроенных функциях рассказано здесь

Функции-расширения (*Extension Functions*)

О расширениях подробно написано в отдельной статье

Высокоуровневые функции и лямбды

О лямбдах и высокоуровневых функциях см. раздел Лямбды

Функции с хвостовой рекурсией

Kotlin поддерживает такой стиль функционального программирования, более известный как "хвостовая рекурсия". Это позволяет использовать циклические алгоритмы вместо рекурсивных функции, но без риска переполнения стэка. Когда функция помечена модификатором `tailrec` и её форма отвечает требованиям компилятора, он

оптимизирует рекурсию, оставляя вместо неё быстрое и эффективное решение этой задачи, основанное на циклах.

```
tailrec fun findFixPoint(x: Double = 1.0): Double
        = if (x == Math.cos(x)) x else findFixPoint(Math.cos(x))
```

Этот код высчитывает `fixpoint` косинуса, который является математической константой. Он просто напросто постоянно вызывает Math.cos, начиная с 1.0 до тех пор, пока результат не изменится, приняв значение 0.7390851332151607. Получившийся код эквивалентен вот этому более традиционному стилю:

```
private fun findFixPoint(): Double {
    var x = 1.0
    while (true) {
        val y = Math.cos(x)
        if (x == y) return y
        x = y
    }
}
```

Для соответствия требованиям модификатора `tailrec`, функция должна вызывать сама себя в качестве последней операции, которую она предпринимает. Вы не можете использовать хвостовую рекурсию, когда существует ещё какой-то код после вызова этой самой рекурсии. Также нельзя использовать её внутри блоков try/catch/finally. На данный момент, хвостовая рекурсия поддерживается только в backend виртуальной машины **Java**(**JVM**).

Высокоуровневые функции и лямбды

Функции высшего порядка

Высокоуровневая функция - это функция, которая принимает другую функцию в качестве входного аргумента, либо имеет функцию в качестве возвращаемого результата. Хорошим примером такой функции является `lock()`, которая берёт залоченный объект и функцию, применяет лок, выполняет функцию и отпускает `lock`:

```
fun <T> lock(lock: Lock, body: () -> T): T{
  lock.lock()
  try{
      return body()
```

```
    }
    finally {
        lock.unlock()
    }
}
```

Давайте проанализируем этот блок. Параметр `body` имеет функциональный тип: `() -> T`, то есть предполагается, что это функция, которая не имеет никаких входных аргументов и возвращает значение типа `T`. Она вызывается внутри блока `try`, защищена `lock`, и её результат возвращается функцией `lock()`.

Если мы хотим вызвать метод lock(), мы можем подать другую функцию в качестве входящего аргумента (более подробно читайте Ссылки на функции):

```
fun toBeSynchronized() = sharedResource.operation()


val result = lock (lock, ::toBeSynchronized)
```

Другой, наиболее удобный способ применения лямбда-выражения:

```
val result = lock(lock, { sharedResource.operation() })
```

Лямбда-выражения более подробно описаны здесь, но в целях продолжить этот раздел, давайте произведём краткий обзор:

- Лямбда-выражения всегда заключены в фигурные скобки,
- Параметры этого выражения (если такие есть) объявлены до знака `->` (параметры могут быть опущены),
- Тело выражения идёт после знака `->`.

В Kotlin существует конвенция, по которой, если последний параметр функции является функцией, и вы применяете лямбда- выражение в качестве аргумента, вы можете указать её вне скобок:

```
lock (lock) {
    sharedResource.operation()
}
```

Другим примером функции высшего порядка служит функция `map()`:

```
fun <T, R> List<T>.map(transform: (T) -> R): List<R> {
    val result = arrayListOf<R>()
    for (item in this)
        result.add(transform(item))
    return result
```

```
}
```

Эта функция может быть вызвана следующим образом:

```
val doubled = ints.map { it -> it * 2 }
```

Обратите внимание, что параметры могут быть проигнорированы при вызове функции в том случае, если лямбда является единственным аргументом для её вызова.

Ключевое слово `it`: неявное имя единственного параметра

Ещё одной полезной особенностью синтаксиса является возможность опустить объявление параметра функции в случае, если он единственный (вместе с `->`).
Слово `it` будет принято в качестве имени для такой функции:

```
ints.map { it * 2 }
```

Это соглашение позволяет писать код в LINQ стиле:

```
strings.filter { it.lenght == 5 }.sortBy { it }.map { it.toUpperCase() }
```

Инлайн функции

Иногда необходимо улучшить производительность высокоуровневых функций, используя инлайн функции.

Лямбда-выражения и анонимные функции

Лямбда-выражения или анонимные функции являются ~~"функциональными константами"~~*(ориг. "functional literal")*, то есть функциями, которые не были объявлены, но сразу были переданы в качестве выражения. Рассмотрим следующий пример:

```
max(strings, { a, b -> a.length < b.length })
```

Функция `max` - высокоуровневая функция, так как она принимает другую функцию в качестве входного аргумента. Этот второй аргумент является выражением, которое само по себе представляет из себя функцию, то есть *functional literal*.

```
fun compare(a: String, b: String): Boolean = a.length < b.length
```

Типы функций

Для того, чтобы функция принимала другую функцию в качестве входного параметра, нам необходимо указать её (входящей функции) тип. К примеру, вышеуказанная функция `max` определена следующим образом:

```
fun <T> max(collection: Collection<T>, less: (T, T) -> Boolean): T? {
    var max: T? = null
    for (it in collection)
        if (max == null || less(max, it))
            max = it
    return max
}
```

Параметр 'less' является `(T, T) -> Boolean` типом, то есть функцией, которая принимает два параметра типа `T` и возвращает 'Boolean':'true', если первый параметр меньше, чем второй.

В теле функции, строка 4, `less` используется в качестве функции: она вызывается путём передачи двух аргументов типа `T`.

Тип функции может быть написан так, как указано выше, или же может иметь определённые параметры, если вы хотите обозначить значения каждого из параметров.

```
val compare: (x: T, y: T) -> Int = ...
```

Синтаксис лямбда-выражений

Полная синтаксическая форма лямбда-выражений, таких как *literals of function types*, может быть представлена следующим образом:

```
val sum = { x: Int, y: Int -> x + y }
```

Лямбда-выражение всегда заключено в скобки `{...}`, объявление параметров при таком синтаксисе происходит внутри этих скобок и может включать в себя аннотации типов (опционально), тело функции начинается после знака `->`. Если тип возвращаемого значения не `Unit`, то в качестве возвращаемого типа принимается последнее (а возможно и единственное) выражение внутри тела лямбды.

Если мы вынесем все необязательные объявления, то, что останется, будет выглядеть следующим образом:

```
val sum: (Int, Int) -> Int = { x, y -> x + y }
```

Обычное дело, когда лямбда-выражение имеет только один параметр. Если **Kotlin** может определить сигнатуру метода сам, он позволит нам не объявлять этот единственный параметр, и объявит его сам под именем `it`:

```
ints.filter { it > 0 } //Эта константа имеет тип '(it: Int) -> Boolean'
```

Мы можем явно вернуть значение из лямбды, используя qualified return синтаксис:

```
ints.filter {
    val shouldFilter = it > 0
```

```
    shouldFilter
}


ints.filter {
    val shouldFilter = it > 0
    return@filter shouldFilter
}
```

Обратите внимение, что функция принимает другую функцию в качестве своего последнего параметра, аргумент лямбда-выражения в таком случае может быть принят вне списка аргументов, заключённого в скобках. См. callSuffix.

Анонимные функции

Единственной особенностью синтаксиса лямбда-выражений, о которой ещё не было сказано, является способность определять и назначать возвращаемый функцией тип. В большинстве случаев в этом нет особой необходимости, потому что он может быть вычислен автоматически. Однако, если у вас есть потребность в определении возвращаемого типа, вы можете воспользоваться альтернативным синтаксисом:

```
fun(x: Int, y: Int): Int = x + y
```

Объявление анонимной функции выглядит очень похоже на обычное объявление функции, за исключением того, что её имя опущено. Тело такой функции может быть описано и выражением (как показано выше), и блоком:

```
fun(x: Int, y: Int): Int {
    return x + y
}
```

Параметры функции и возвращаемый тип обозначаются таким же образом, как в обычных функциях. Правда, тип параметра может быть опущен, если его значение следует из контекста:

```
ints.filter(fun(item) = item > 0)
```

Аналогично и с типом возвращаемого значения: он вычисляется автоматически для функций-выражений или же должен быть определён вручную (если не является типом `Unit`) для анонимных функций, которые имеют в себе блок.

Обратите внимание, что параметры анонимных функций всегда заключены в скобки `{...}`. Приём, позволяющий оставлять параметры вне скобок, работает только с лямбда-выражениями.

Одним из отличий лямбда-выражений от анонимных функций является поведение оператора `return`(non-local returns). Слово `return` , не имеющее метки (`@`), всегда возвращается из функции, объявленной ключевым словом `fun`. Это означает,

что `return` внутри лямбда-выражения возвратит выполнение к функции, включающей в себя это лямбда-выражение. Внутри анонимных функций оператор `return`, в свою очередь, выйдет, собственно, из анонимной функции.

Замыкания

Лямбда-выражение или анонимная функция (так же, как и локальная функция или object expression) имеет доступ к своему замыканию, то есть к переменным, объявленным вне этого выражения или функции. В отличае от Java, переменные, захваченные в замыкании, могут быть изменены:

```
var sum = 0
ints.filter { it > 0 }.forEach {
    sum += it
}
print(sum)
```

Литералы функций с объектом-приёмником

Kotlin предоставляет возможность вызывать литерал функции с указаным объектом-приёмником. Внутри тела литерала вы можете вызывать методы объекта-приёмника без дополнительных определителей. Это схоже с принципом работы расширений, которые позволяют получить доступ к членам объекта-приёмника внутри тела функции. Один из самых важных примеров использования литералов с объектом-приёмником это Type-safe Groovy-style builders.

Тип такого литерала — это тип функции с приёмником:

```
sum : Int.(other: Int) -> Int
```

По аналогии с расширениями, литерал функции может быть вызван так, будто он является методом объекта-приёмника:

```
1.sum(2)
```

Синтаксис анонимной функции позволяет вам явно указать тип приёмника. Это может быть полезно в случае, если вам нужно объявить переменную типа нашей функции для использования в дальнейшем.

```
val sum = fun Int.(other: Int): Int = this + other
```

Лямбда-выражения могут быть использованы как литералы функций с приёмником, когда тип приёмника может быть выведен из контекста.

```
class HTML {
    fun body() { ... }
```

```
}


fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()  // создание объекта-приёмника
    html.init()        // передача приёмника в лямбду
    return html
}



html {       // лямбда с приёмником начинается тут
    body()   // вызов метода объекта-приёмника
}
```

Встроенные (inline) функции

Использование функций высшего порядка влечёт за собой снижение производительности: во-первых, функция является объектом, а во-вторых, происходит захват контекста замыканием, то есть функции становятся доступны переменные, объявленные вне её тела. А выделения памяти (как для объекта функции, так и для её класса) и виртуальные вызовы занимают системные ресурсы.

Но во многих случаях эти "накладные расходы" можно устранить с помощью инлайнинга (встраивания) лямбда-выражений. Например, функция `lock()` может быть легко встроена в то место, из которого она вызывается:

```
lock(l) { foo() }
```

Вместо создания объекта функции для параметра и генерации вызова, компилятор мог бы выполнить что-то подобное этому коду:

```
l.lock()
try {
    foo()
}
finally {
    l.unlock()
}
```

Разве это не то, чего мы хотели изначально?

Чтобы заставить компилятор поступить именно так, нам необходимо отметить функцию `lock`модификатором `inline`:

```
inline fun lock<T>(lock: Lock, body: () -> T): T {
    // ...
}
```

Модификатор `inline` влияет и на функцию, и на лямбду, переданную ей: они обе будут встроены в место вызова.

Встраивание функций может увеличить количество сгенерированного кода, но если вы будете делать это в разумных пределах (не инлайнить большие функции), то получите прирост производительности, особенно при вызове функций с параметрами разного типа внутри циклов.

noinline

В случае, если вы хотите, чтобы только некоторые лямбды, переданные inline-функции, были встроены, вам необходимо отметить модификатором `noinline` те функции-параметры, которые встроены не будут:

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) {
    // ...
}
```

Когда как встраиваемые лямбды могут быть вызваны только внутри inline-функций или переданы в качестве встраиваемых аргументов, с `noinline`-функциями можно работать без ограничений: хранить внутри полей, передавать куда-либо и т.д.

Заметьте, что если inline-функция не имеет ни inline параметров, ни параметров вещественного типа, компилятор выдаст предупреждение, так как встраивание такой функции вряд ли принесёт пользу (вы можете скрыть предупреждение, если уверены, что встраивание необходимо).

Нелокальные return

В Kotlin мы можем использовать обыкновенный, безусловный `return` только для выхода из именованной функции или анонимной функции. Это значит, что для выхода из лямбды нам нужно использовать label. Обычный `return` запрещён внутри лямбды, потому что она не может заставить внешнюю функцию завершиться.

```
fun foo() {
    ordinaryFunction {
        return // ERROR: can not make `foo` return here
    }
}
```

Но если функция, в которую передана лямбда, встроена, то `return` также будет встроен, поэтому так делать можно:

```
fun foo() {
    inlineFunction {
        return // OK: the lambda is inlined
    }
}
```

Такие return (находящиеся внутри лямбд, но завершающие внешнюю функцию) называются нелокальными (`non-local`). Мы используем такие конструкции в циклах, которые являются inline-функциями:

```
fun hasZeros(ints: List<Int>): Boolean {
    ints.forEach {
        if (it == 0) return true // returns from hasZeros
    }
    return false
}
```

Заметьте, что некоторые inline-функции могут вызывать переданные им лямбды не напрямую в теле функции, а из иного контекста, такого как локальный объект или вложенная функция. В таких случаях, нелокальное управление потоком выполнения также запрещено в лямбдах. Чтобы указать это, параметр лямбды необходимо отметить модификатором `crossinline`:

```
inline fun f(crossinline body: () -> Unit) {
    val f = object: Runnable {
        override fun run() = body()
    }
    // ...
}
```

`break` и `continue` пока что недоступны во встроенных лямбдах, но мы планируем добавить их поддержку

Параметры вещественного типа

Иногда нам необходимо получить доступ к типу, переданному в качестве параметра:

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
```

```
    while (p != null && !clazz.isInstance(p)) {
        p = p?.parent
    }
    @Suppress("UNCHECKED_CAST")
    return p as T
}
```

В этом примере мы осуществляем проход по дереву и используем рефлексию, чтобы проверить узел на принадлежность к определённому типу. Это прекрасно работает, но вызов выглядит не очень симпатично:

```
myTree.findParentOfType(MyTreeNodeType::class.java)
```

Что мы на самом деле хотим, так это передать этой функции тип, то есть вызвать её вот так:

```
myTree.findParentOfType<MyTreeNodeType>()
```

В таких случаях inline-функции могут принимать *параметры вещественного типа* (reified type parameters). Чтобы включить эту возможность, мы можем написать что-то вроде этого:

```
inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p?.parent
    }
    return p as T
}
```

Мы определили тип параметра с помощью модификатора `reified`, но он доступен внутри функции почти так же, как и обычный класс. Так как функция встроена, то для работы таких операторов как `!is` и `as` рефлексия не нужна. Также, мы можем вызывать её таким же образом, как было упомянуто выше: `myTree.findParentOfType<MyTreeNodeType>()`

Хотя рефлексия может быть не нужна во многих случаях, мы всё ещё можем использовать её с параметром вещественного типа:

```
inline fun <reified T> membersOf() = T::class.members

fun main(s: Array<String>) {
    println(membersOf<StringBuilder>().joinToString("\n"))
}
```

Обычная функция (не отмеченная как встроенная) не может иметь параметры вещественного типа. Тип, который не имеет представление во времени исполнения (например, параметр невещественного или фиктивного типа вроде `Nothing`), не может использоваться в качестве аргумента для параметра вещественного типа.

Для низкоуровневого описания см. спецификацию.

Сопрограммы

Сопрограммы являются *экспериментальными* в Kotlin 1.1. Детали см. ниже

Некоторые API инициируют долго протекающие операции (такие как сетевой ввод-вывод, файловый ввод-вывод, интенсивная обработка на CPU или GPU и др.), которые требуют блокировки вызывающего кода в ожидании завершения операций. Сопрограммы обеспечивают возможность избежать блокировки исполняющегося потока путём использования более дешёвой и управляемой операции: *приостановки*(suspend) сопрограммы.

Сопрограммы упрощают асинхронное программирование, оставив все осложнения внутри библиотек. Логика программы может быть выражена *последовательно* в сопрограммах, а базовая библиотека будет её реализовывать асинхронно для нас. Библиотека может обернуть соответствующие части кода пользователя в обратные вызовы (callbacks), подписывающиеся на соответствующие события, и диспетчировать исполнение на различные потоки (или даже на разные машины!). Код при этом останется столь же простой, как если бы исполнялся строго последовательно.

Многие асинхронные механизмы, доступные в других языках программирования, могут быть реализованы в качестве библиотек с помощью сопрограмм Kotlin. Это включает в себя `async`/`await` из C# и ECMAScript, channels и `select` из языка Go, и generators /`yield` из C# или Python. См. описания ниже о библиотеках, реализующих такие конструкции.

Блокирование против приостановки

Главным отличительным признаком сопрограмм является то, что они являются вычислениями, которые могут быть *приостановлены без блокирования потока* (вытеснения средствами операционной системы). Блокирование потоков часто является весьма дорогостоящим, особенно при интенсивных нагрузках: только относительно небольшое число потоков из общего числа является активно выполняющимися, поэтому блокировка одного из них ведет к затягиванию какой-нибудь важной части итоговой работы.

С другой стороны, приостановка сопрограммы обходится практически бесплатно. Не требуется переключения контекста (потоков) или иного вовлечения механизмов операционной системы. И сверх этого, приостановка может гибко контролироваться пользовательской библиотекой во многих аспектах: в качестве авторов библиотеки мы можем решать, что происходит при приостановке, и оптимизировать, журналировать или перехватывать в соответствии со своими потребностями.

Еще одно отличие заключается в том, что сопрограммы не могут быть приостановлены на произвольной инструкции, а только в так называемых *точках остановки* (приостановки), которые вызываются в специально маркируемых функциях.

Останавливаемые функции

Приостановка происходит в случае вызова функции, обозначенной специальным модификатором `suspend`:

```
suspend fun doSomething(foo: Foo): Bar {
    ...
}
```

Такие функции называются *функциями остановки* (приостановки), поскольку их вызовы могут приостановить выполнение сопрограммы (библиотека может принять решение продолжать работу без приостановки, если результат вызова уже доступен). Функции остановки могут иметь параметры и возвращать значения точно так же, как и все обычные функции, но они могут быть вызваны только из сопрограмм или других функций остановки. В конечном итоге, при старте сопрограммы она должна содержать как минимум одну функцию остановки, и функция эта обычно анонимная (лямбда-функция остановки). Давайте взглянем, для примера, на упрощённую функцию `async()` (из библиотеки `kotlinx.coroutines`):

```
fun <T> async(block: suspend () -> T)
```

Здесь `async()` является обычной функцией (не функцией остановки), но параметр `block` имеет функциональный тип с модификатором `suspend`: `suspend () -> T`. Таким образом, когда мы передаём лямбда-функцию в `async()`, она является анонимной функцией остановки, и мы можем вызывать функцию остановки изнутри её:

```
async {
    doSomething(foo)
    ...
}
```

Продолжая аналогию, `await()` может быть функцией остановки (также может вызываться из блока `async {}`), которая приостанавливает сопрограмму до тех пор, пока некоторые вычисления не будут выполнены, и затем возвращает их результат:

```
async {
    ...
    val result = computation.await()
    ...
}
```

Больше информации о том, как действительно работают функции `async/await` в`kotlinx.coroutines`, может быть найдено здесь.

Отметим, что функции приостановки `await()` и `doSomething()` не могут быть вызваны из обыкновенных функций, подобных `main()`:

```
fun main(args: Array<String>) {
    doSomething() // ERROR: Suspending function called from a non-coroutine context
}
```

Заметим, что функции остановки могут быть виртуальными, и при их переопределении модификатор `suspend` также должен быть указан:

```
interface Base {
    suspend fun foo()
}

class Derived: Base {
    override suspend fun foo() { ... }
}
```

Аннотация @RestrictsSuspension

Расширяющие функции (и анонимные функции) также могут быть маркированы как `suspend`, подобно и всем остальным (регулярным) функциям. Это позволяет создавать DSL и другие API, которые пользователь может расширять. В некоторых случаях автору библиотеки необходимо запретить пользователю добавлять *новые пути* приостановки сопрограммы.

Чтобы осуществить это, можно использовать аннотацию `@RestrictsSuspension`. Когда целевой класс или интерфейс `R` аннотируется подобным образом, все расширения приостановки должны делегироваться либо из членов `R`, либо из других его расширений. Поскольку расширения не могут делегировать друг друга до бесконечности (иначе программа никогда не завершится), гарантируется, что все приостановки пройдут посредством вызова члена `R`, так что автор библиотеки может полностью их контролировать.

Это актуально в тех *редких* случаях, когда каждая приостановка обрабатывается специальным образом в библиотеке. Например, при реализации генераторов через `buildSequence()` функцию, описанную ниже, мы должны быть уверены, что любой приостанавливаемый вызовов в сопрограмме завершается вызовом либо `yield()`, либо `yieldAll()`, а не какой-либо другой функции. Именно по этой причине `SequenceBuilder` аннотирована с @RestrictsSuspension:

```
@RestrictsSuspension
public abstract class SequenceBuilder<in T> {
```

```
    ...
}
```

См. исходники на Github.

Внутреннее функционирование сопрограмм

Мы не стремимся здесь дать полное объяснение того, как сопрограммы работают под капотом, но примерный смысл того, что происходит, очень важен.

Сопрограммы полностью реализованы с помощью технологии компиляции (поддержка от языковой виртуальной машины, среды исполнения, или операционной системы не требуется), а приостановка работает через преобразование кода. В принципе, каждая функция приостановки (оптимизации могут применяться, но мы не будем вдаваться в эти подробности здесь) преобразуется в конечный автомат, где состояния соответствуют приостановленным вызовам. Прямо перед приостановкой следующее состояние загружается в поле сгенерированного компилятором класса вместе с сопутствующими локальным переменными и т. д. При возобновлении сопрограммы локальные переменные и состояние восстанавливаются, и конечный автомат продолжает свою работу.

Приостановленную сопрограмму можно сохранять и передавать как объект, который хранит её приостановленное состояние и локальные переменные. Типом таких объектов является Continuation, а преобразование кода, описанное здесь, соответствует классическому Continuation-passing style. Следовательно, приостановливаемые функции принимают дополнительный параметр типа `Continuation` (сохранённое состояние) под капотом.

Более детально о том, как работают сопрограммы, можно узнать в этом проектном документе. Похожие описания `async / await` в других языках (таких как C# или ECMAScript 2016) актуальны и здесь, хотя особенности их языковых реализаций могут существенно отличаться от сопрограмм Kotlin.

Экспериментальный статус сопрограмм

Дизайн сопрограмм носит статус experimental, из чего следует возможность его изменения в будущих релизах. При составлении сопрограммы в Kotlin 1.1 по умолчанию выводится предупреждение: *The feature "coroutines" is experimental*. Чтобы убрать предупреждение, необходимо указать опцию opt-in flag.

Из-за экспериментального статуса сопрограмм все связанные API собраны в стандартной библиотеке как пакет `kotlin.coroutines.experimental`. Когда дизайн будет стабилизирован и его экспериментальный статус снят, окончательный API будет перенесен в пакет `kotlin.coroutines`, а экспериментальный пакет будет храниться (возможно, как отдельный артефакт) в целях обеспечения обратной совместимости.

Важное замечание: мы рекомендуем авторам библиотек следовать той же конвенции: добавить к названию суффикс «экспериментальный»
(например, `com.example.experimental`), указывающий, какой там используется сопрограммно совместимый API. Таким образом ваша библиотека сохранит бинарную совместимость. А когда выйдет финальный API-интерфейс, выполните следующие действия:

- скопируйте все API в `com.example` (без experimental суффикса);
- сохраните экспериментальный вариант пакета для обратной совместимости.

Это позволит минимизировать проблемы миграции для пользователей.

Стандартные API

Сопрограммы представлены в трёх их главных ингредиентах:

- языковая поддержка (функции остановки, как описывалось выше),
- низкоуровневый базовый API в стандартной библиотеке Kotlin,
- API высокого уровня, которые могут быть использованы непосредственно в пользовательском коде.

Низкий уровень API: kotlin.coroutines

Низкоуровневый API относительно мал и должен использоваться ТОЛЬКО для создания библиотек высокого уровня. Он содержит два главных пакета:

- `kotlin.coroutines.experimental` - главные типы и примитивы, такие как:
  - `createCoroutine()`
  - `startCoroutine()`
  - `suspendCoroutine()`
- `kotlin.coroutines.experimental.intrinsics` - встроенные функции еще более низкого уровня, такие как `suspendCoroutineOrReturn`

Более детальная информация о использовании этих API может быть найдена здесь.

API генераторов в kotlin.coroutines

Это функции исключительно «уровня приложения»
в `kotlin.coroutines.experimental`:

- `buildSequence()`
- `buildIterator()`

Они перенесены в рамки `kotlin-stdlib`, поскольку они относятся к последовательностям. По сути, эти функции (и мы можем ограничиться здесь рассмотрением только `buildSequence()`) реализуют генераторы, т. е. предоставляют лёгкую возможность построить ленивые последовательности:

```
import kotlin.coroutines.experimental.*
```

```
fun main(args: Array<String>) {
//sampleStart
    val fibonacciSeq = buildSequence {
        var a = 0
        var b = 1

        yield(1)

        while (true) {
            yield(a + b)

            val tmp = a + b
            a = b
            b = tmp
        }
    }
//sampleEnd

    // Print the first five Fibonacci numbers
    println(fibonacciSeq.take(8).toList())
}
```

Это сгенерирует ленивую, потенциально бесконечную последовательность Фибоначчи, используя сопрограмму, которая дает последовательные числа Фибоначчи, вызывая функцию yield (). При итерировании такой последовательности на каждом шаге итератор выполняет следующую часть сопрограммы, которая генерирует следующее число. Таким образом, мы можем взять любой конечный список чисел из этой последовательности, например `fibonacciSeq.take(8).toList()`, дающий в результате `[1, 1, 2, 3, 5, 8, 13, 21]`. И сопрограммы достаточно дёшевы, чтобы сделать это практичным.

Чтобы продемонстрировать реальную ленивость такой последовательности, давайте напечатаем некоторые отладочные результаты изнутри вызова buildSequence():

```
import kotlin.coroutines.experimental.*

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

```
//sampleStart
    val lazySeq = buildSequence {
        print("START ")
        for (i in 1..5) {
            yield(i)
            print("STEP ")
        }
        print("END")
    }

    // Print the first three elements of the sequence
    lazySeq.take(3).forEach { print("$it ") }
//sampleEnd
}
```

Запустите приведенный выше код, чтобы убедиться, что если мы будем печатать первые три элемента, цифры чередуются со `STEP`-ами по ветвям цикла. Это означает, что вычисления действительно ленивые. Для печати `1` мы выполняем только до первого `yield(i)` и печатаем `START` ходу дела. Затем, для печати `2`, нам необходимо переходить к следующему `yield(i)`, и здесь печатать `STEP`. То же самое и для `3`. И следующий `STEP` никогда не будет напечатан (точно так же как и `END`), поскольку мы никогда не запрашиваем дополнительных элементов последовательности.

Чтобы сразу породить всю коллекцию (или последовательность) значений, доступна функция `yieldAll()`:

```
import kotlin.coroutines.experimental.*

fun main(args: Array<String>) {
//sampleStart
    val lazySeq = buildSequence {
        yield(0)
        yieldAll(1..10)
    }

    lazySeq.forEach { print("$it ") }
//sampleEnd
```

```
}
```

Функция `buildIterator()` во всём подобна buildSequence(), но только возвращает ленивый итератор.

Вы могли бы добавить собственную логику выполнения функции `buildSequence()`, написав приостанавливаемое расширение класса `SequenceBuilder` (что порождается аннотацией `@RestrictsSuspension`, как описывалось выше):

```
import kotlin.coroutines.experimental.*


//sampleStart
suspend fun SequenceBuilder<Int>.yieldIfOdd(x: Int) {
    if (x % 2 != 0) yield(x)
}


val lazySeq = buildSequence {
    for (i in 1..10) yieldIfOdd(i)
}
//sampleEnd


fun main(args: Array<String>) {
    lazySeq.forEach { print("$it ") }
}
```

Другие API высокого уровня: kotlinx.coroutines

Только базовые API, связанные с сопрограммами, доступны непосредственно из стандартной библиотеки Kotlin. Они преимущественно состоят из основных примитивов и интерфейсов, которые, вероятно, будут использоваться во всех библиотеках на основе сопрограмм.

Большинство API уровня приложений, основанные на сопрограммах, реализованы в отдельной библиотеке `kotlinx.coroutines`. Эта библиотека содержит в себе:

- Платформенно-зависимое асинхронное программирование с помощью `kotlinx-coroutines-core`:
  - этот модуль включает Go-подобные каналы, которые поддерживают `select` и другие удачные примитивы
  - исчерпывающее руководство по этой библиотеке доступно здесь.
- API, основанные на `CompletableFuture` из JDK 8: `kotlinx-coroutines-jdk8`
- Неблокирующий ввод-вывод (NIO), основанный на API из JDK 7 и выше: `kotlinx-coroutines-nio`

- Поддержка Swing (`kotlinx-coroutines-swing`) и JavaFx (`kotlinx-coroutines-javafx`)
- Поддержка RxJava: `kotlinx-coroutines-rx`

Эти библиотеки являются удобными API, которые делают основные задачи простыми. Также они содержат законченные примеры того, как создавать библиотеки, построенные на сопрограммах.

Мульти-декларации

Иногда удобно *деструктуризировать* объект на несколько переменных, например:

```
val (name, age) = person
```

Этот синтаксис называется *деструктуризирующее присваивание*. Он позволяет присвоить объект сразу нескольким переменным, разбив его на части. Мы объявили две переменные: `name` и `age`, и теперь можем использовать их по отдельности:

```
println(name)

println(age)
```

Эта декларация транслируется в такой код:

```
val name = person.component1()

val age = person.component2()
```

Как и многое другое в Kotlin, мульти-декларации опираются на конвенцию: функции `componentN()` вызываются по имени, то есть могут быть объявлены как в классе person, так и вне его — в качестве расширений.

Заметьте, что функции `componentN()` нужно отмечать ключевым словом `operator`, чтобы позволить их использование в деструктуризирующем присваивании.

Деструктуризирующие присваивания также работают в циклах `for`:

```
for ((a, b) in collection) { ... }
```

В данном примере значения переменных `a` и `b` возращены методами `component1()` и `component2()`, вызванными неявно у элементов коллекции.

Например: возврат двух значений из функции

Предположим, нам нужно вернуть два значения из функции. Например, результат вычисления и какой-нибудь статус. Компактный способ достичь этого — объявление `data`-класса и возвращение его экземпляра:

```
data class Result(val result: Int, val status: Status)
```

```
fun function(...): Result {
    // вычисления

    return Result(result, status)
}

// Теперь мы можем использовать деструктуризирующее присваивание:
val (result, status) = function(...)
```

Так как `data`-классы автоматически объявляют `componentN()`-функции, мульти-декларации будут работать с ними "из коробки".

ПРИМЕЧАНИЕ: мы также могли использовать стандартный класс `Pair`, чтобы заставить функцию вернуть `Pair<Int, Status>`, но правильнее будет именовать ваши данные должным образом.

Пример: мульти-декларации и ассоциативные списки

Пожалуй, самый хороший способ итерации по ассоциативному списку:

```
for ((key, value) in map) {
   // do something with the key and the value
}
```

Чтобы это работало, мы должны:

- представить ассоциативный список как последовательность значений, предоставив функцию `iterator()`,
- представить каждый элемент как пару с помощью функций `component1()` и `component2()`.

И да, стандартная библиотека предоставляет такие расширения:

```
operator fun <K, V> Map<K, V>.iterator(): Iterator<Map.Entry<K, V>> = entr
ySet().iterator()

operator fun <K, V> Map.Entry<K, V>.component1() = getKey()

operator fun <K, V> Map.Entry<K, V>.component2() = getValue()
```

Так что вы можете свободно использовать мульти-декларации в циклах `for` с ассоциативными списками (так же как и с коллекциями экземпляров `data`-классов).

Коллекции

В отличие от многих языков, Kotlin различает изменяемые и неизменяемые коллекции (списки, множества, ассоциативные списки и т.д.). Точный контроль над тем, когда именно коллекции могут быть изменены, полезен для устранения багов и разработки хорошего API.

Важно понимать различие между read-only *представлением* изменяемой коллекции, и фактически неизменяемой коллекцией. Их легко создать, но вот система типов не выражает различие между ними, поэтому следить за этим должны вы (если это необходимо).

Тип `List<out T>` в Kotlin — интерфейс, который предоставляет read-only операции, такие как `size`, `get`, и другие. Так же, как и в Java, он наследуется от `Collection<T>`, а значит и от `Iterable<T>`. Методы, которые изменяют список, добавлены в интерфейс `MutableList<T>`. То же самое относится и к `Set<out T>/MutableSet<T>`, `Map<K, out V>/MutableMap<K, V>`.

Пример базового использования списка (list) и множества (set):

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
val readOnlyView: List<Int> = numbers
println(numbers)        // выведет "[1, 2, 3]"
numbers.add(4)
println(readOnlyView)   // выведет "[1, 2, 3, 4]"
readOnlyView.clear()    // -> не скомпилируется


val strings = hashSetOf("a", "b", "c", "c")
assert(strings.size == 3)
```

Kotlin не имеет специальных синтаксических конструкций для создания списков или множеств. Используйте методы из стандартной библиотеки, такие как `listOf()`, `mutableListOf()`, `setOf()`, `mutableSetOf()`. Создание ассоциативного списка некритичным для производительности способом может быть осуществленно с помощью простой идиомы: `mapOf(a to b, c to d)`

Заметьте, что переменная `readOnlyView` изменяется вместе со списком, на который она указывает. Если единственная ссылка, указывающая на список является read-only, то мы можем с уверенностью сказать, что коллекция полностью неизменяемая. Простой способ создания такой коллекции выглядит так:

```
val items = listOf(1, 2, 3)
```

В данный момент, метод `listOf` реализован с помощью ArrayList, но не исключено, что в будущем могут быть использованы другие типы коллекций, более эффективные по памяти за счёт своей неизменности.

Заметьте, что read-only типы ковариантны. Это значит, что вы можете взять `List<Rectangle>` (список прямоугольников) и присвоить его `List<Shape>` (списку фигур) предполагая, что Rectangle наследуется от Shape. Такое присвоение было бы запрещено с изменяемыми коллекциями, потому что в таком случае появляется риск возникновения ошибок времени исполнения.

Иногда вам необходимо вернуть состояние коллекции в определённый момент времени:

```
class Controller {

    private val _items = mutableListOf<String>()

    val items: List<String> get() = _items.toList()

}
```

Расширение `toList` просто копирует элементы списка. Таким образом, возвращаемый список гарантированно не изменится.

Существует несколько полезных расширений для списков и множеств, с которыми стоит познакомиться:

```
val items = listOf(1, 2, 3, 4)

items.first() == 1

items.last() == 4

items.filter { it % 2 == 0 }   // возвратит [2, 4]


val rwList = mutableListOf(1, 2, 3)

rwList.requireNoNulls()        // возвратит [1, 2, 3]

if (rwList.none { it > 6 }) println("Нет элементов больше 6")  // выведет "Нет элементов больше 6"

val item = rwList.firstOrNull()
```

Также, обратите внимание на такие утилиты как `sort`, `zip`, `fold`, `reduce`.

То же самое происходит и с ассоциативными списками. Они могут быть с лёгкостью инициализированы и использованы следующим образом:

```
val readWriteMap = hashMapOf("foo" to 1, "bar" to 2)

println(readWriteMap["foo"])  // выведет "1"

val snapshot: Map<String, Int> = HashMap(readWriteMap)
```

Интервалы

Интервалы оформлены с помощью функций `rangeTo` и имеют оператор в виде `..`, который дополняется *in* и *!in*. Они применимы ко всем сравниваемым (*comparable*) типам,

но для целочисленных примитивов есть оптимизированная реализация. Вот несколько примеров применения интервалов.

```
if (i in 1..10) { // equivalent of 1 <= i && i <= 10
    println(i)
}
```

Интервалы целочисленного типа (`IntRange`, `LongRange`, `CharRange`) имеют определённое преимущество: они могут иметь дополнительную итерацию. Компилятор конвертирует такие интервалы в аналогичные циклы *for* из языка **Java**.

```
for (i in 1..4) print(i) // prints "1234"

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

А что, если вы хотите произвести итерацию в обратном порядке? Это просто. Можете использовать функцию `downTo()`, определённую в стандартной библиотеке:

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

А есть ли возможность производить итерацию с шагом, отличным от единицы? Разумеется. В этом вам поможет функция `step()`:

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

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

Для создания интервала, который не включает последний элемент перебора, используйте `until`:

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

Как это работает

Интервалы реализуют интерфейс `ClosedRange<T>`.

Говоря математическим языком, интерфейс `ClosedRange<T>` обозначет ограниченный отрезок и предназначен для типов, подлежащих сравнению. У него есть две контрольные точки: `start` и `endInclusive`. Главной операцией является `contain`. Чаще всего она используется вместе с операторами **in**/**!in**.

Целочисленные последовательности (`IntProgression`, `LongProgression`, `CharProgression`) являются арифметическими. Последовательности определены элементами `first`, `last` и ненулевым значением `increment`. Элемент `first` является первым, последующими являются элементы, полученные при инкрементации предыдущего элемента с

помощью `increment`. Если последовательность не является пустой, то элемент `last` всегда достигается в результате инкрементации.

Последовательность является подтипом `Iterable<N>`, где `N` - это `Int`, `Long` или `Char`. Таким образом, её можно использовать в циклах **for** и функциях типа `map`, `filter` и т.п. Итерация `Progression` идентична индексируемому циклу **for** в **Java/JavaScript**

```
for (int i = first; i != last; i += increment) {
  // ...
}
```

Для целочисленных типов оператор `..` создаёт объект, который реализует в себе `ClosedRange<T>` и `*Progression*`. К примеру, `IntRange` наследуется от класса `IntProgression` и реализует интерфейс `ClosedRange<Int>`. Поэтому все операторы, обозначенные для `IntProgression`, также доступны и для`IntRange`. Результатом функций `downTo()` и `step()` всегда будет `*Progression*`(перев.: *последовательность*).

Последовательности спроектированы с использованием функции `fromClosedRange` в их вспомогательном объекте (*companion object*):

```
    IntProgression.fromClosedRange(start, end, increment)
```

Для нахождения максимального значения в прогрессии вычисляется элемент `last`. Для последовательности с положительным инкрементом этот элемент вычисляется так, чтобы он был не больше элемента `end`. Для тех последовательностей, где инкремент отрицательный - не меньше.

Вспомогательные функции (ориг.: *Utility functions*)

`rangeTo()`

Операторы `rangeTo()` для целочисленных типов просто вызывают конструктор класса `*Range*`:

```
class Int {
    //...
    operator fun rangeTo(other: Long): LongRange = LongRange(this, other)
    //...
    operator fun rangeTo(other: Int): IntRange = IntRange(this, other)
    //...
}
```

Числа с плавающей точкой (`Double`, `Float`) не имеют своего оператора `rangeTo`. Такой оператор обозначен для них в дженериках типа `Comparable` стандартной библиотеки:

```
    public operator fun <T: Comparable<T>> T.rangeTo(that: T): ClosedRange
<T>
```

Интервал, полученный с помощью такой функции, не может быть использован для итерации.

`downTo()`

Экстеншн-функция `downTo()` задана для любой пары целочисленных типов, вот два примера:

```
fun Long.downTo(other: Int): LongProgression {
    return LongProgression.fromClosedRange(this, other, -1.0)
}


fun Byte.downTo(other: Int): IntProgression {
    return IntProgression.fromClosedRange(this, other, -1)
}
```

`reversed()`

Функция `reversed()` расширяет класс `*Progression*` таким образом, что все экземпляры этого класса возвращают обратные последовательности при её вызове.

```
fun IntProgression.reversed(): IntProgression {
    return IntProgression.fromClosedRange(last, first, -increment)
}
```

`step()`

Функция-расширение `step()` также определена для классов `*Progression*`. Она возвращает последовательность с изменённым значением шага `step` (параметр функции). Значение шага всегда должно быть положительным числом для того, чтобы функция никогда не меняла направления своей итерации.

```
fun IntProgression.step(step: Int): IntProgression {
    if (step <= 0) throw IllegalArgumentException("Step must be positive,
was: $step") //шаг должен быть положительным
    return IntProgression.fromClosedRange(first, last, if (increment > 0)
step else -step)
}
```

```
fun CharProgression.step(step: Int): CharProgression {

    if (step <= 0) throw IllegalArgumentException("Step must be positive,
was: $step")

    return CharProgression.fromClosedRange(first, last, step)

}
```

Обратите внимание, что значение элемента `last` в возвращённой последовательности может отличаться от значения `last` первоначальной последовательности с тем, чтобы предотвратить инвариант `(last – first) % increment == 0`. Вот пример:

```
    (1..12 step 2).last == 11  // последовательность чисел со значениями [
1, 3, 5, 7, 9, 11]

    (1..12 step 3).last == 10  // последовательность чисел со значениями [
1, 4, 7, 10]

    (1..12 step 4).last == 9   // последовательность чисел со значениями [
1, 5, 9]
```

Приведение и проверка типов

Операторы `is` и `!is`

Мы можем проверить принадлежит ли объект к какому-либо типу во время исполнения с помощью оператора `is` или его отрицания `!is`:

```
if (obj is String) {

    print(obj.length)

}


if (obj !is String) { // то же самое, что и !(obj is String)

    print("Not a String")

}

else {

    print(obj.length)

}
```

Умные приведения

Во многих случаях в Kotlin вам не нужно использовать явные приведения, потому что компилятор следит за `is`-проверками для неизменяемых значений и вставляет приведения автоматически, там, где они нужны:

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x автоматически преобразовывается в String
    }
}
```

Компилятор достаточно умён для того, чтобы делать автоматические приведения в случаях, когда проверка на несоответствие типу (`!is`) приводит к выходу из функции:

```
    if (x !is String) return
    print(x.length) // x автоматически преобразовывается в String
```

или в случаях, когда приводимая переменная находится справа от оператора `&&` или `||`:

```
    // x автоматически преобразовывается в String справа от `||`
    if (x !is String || x.length == 0) return


    // x автоматически преобразовывается в String справа от `&&`
    if (x is String && x.length > 0) {
        print(x.length) // x автоматически преобразовывается в String
    }
```

Такие *умные приведения* работают вместе с *when*-выражениями и циклами *while*:

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.length + 1)
    is IntArray -> print(x.sum())
}
```

Заметьте, что умные приведения не работают, когда компилятор не может гарантировать, что переменная не изменится между проверкой и использованием. Более конкретно, умные приведения будут работать:

- с локальными *val* переменными - всегда;
- с *val* свойствами - если поле имеет модификатор доступа `private` или `internal`, или проверка происходит в том же модуле, в котором объявлено это свойство. Умные приведения неприменимы к публичным свойствам или свойствам, которые имеют переопределённые getter'ы;
- с локальными *var* переменными - если переменная не изменяется между проверкой и использованием и не захватывается лямбдой, которая её модифицирует;

- с *var* свойствами - никогда (потому что переменная может быть изменена в любое время другим кодом).

Оператор "небезопасного" приведения

Обычно оператор приведения выбрасывает исключение, если приведение невозможно, поэтому мы называем его *небезопасным*. Небезопасное приведение в Kotlin выполняется с помощью инфиксного оператора *as* (см. приоритеты операторов):

```
val x: String = y as String
```

Заметьте, что *null* не может быть приведен к `String`, так как `String` не является nullable, т.е. если `y` - null, код выше выбросит исключение. Чтобы соответствовать семантике приведений в Java, нам нужно указать nullable тип в правой части приведения:

```
val x: String? = y as String?
```

Оператор "безопасного" (nullable) приведения

Чтобы избежать исключения, вы можете использовать
оператор *безопасного* приведения *as?*, который возвращает *null* в случае неудачи:

```
val x: String? = y as? String
```

Заметьте, что несмотря на то, что справа от *as?* стоит non-null тип `String`, результат приведения является nullable.

Ключевое слово this

Чтобы сослаться на объект, с которым мы работаем, используется ключевое слово *this*:

- Внутри класса ключевое слово *this* ссылается на объект этого класса
- В функциях-расширениях или в литерале функции с принимающим объектом *this* обозначает*принимающий объект*, который передаётся слева от точки.

Если ключевое слово *this* не имеет определителей, то оно ссылается на *область самого глубокого замыкания*. Чтобы сослаться на *this* в одной из внешних областей, используются *метки-определители*:

this с определителем

Чтобы получить доступ к *this* из внешней области (класса, функции-расширения, или именованных литералов функций с принимающим объектом) мы пишем `this@label`, где `@label` - это метка области, из которой нужно получить *this*:

```
class A { // неявная метка @A
    inner class B { // неявная метка @B
        fun Int.foo() { // неявная метка @foo
            val a = this@A // this из A
            val b = this@B // this из B

            val c = this // принимающий объект функции foo(), типа Int
            val c1 = this@foo // принимающий объект функции foo(), типа In
t

            val funLit = lambda@ fun String.() {
                val d = this // принимающий объект литерала funLit
            }

            val funLit2 = { s: String ->
                // принимающий объект функции foo(), т.к. замыкание лямбды
не имеет принимающего объекта
                val d1 = this
            }
        }
    }
}
```

Равенство

В Kotlin есть два типа равенства:

- Равенство ссылок (две ссылки указывают на один и тот же объект)
- Равенство структур (проверка через `equals()`)

Равенство ссылок

Равенство ссылок проверяется с помощью оператора `===` (и его отрицания `!==`). Выражение `a === b` является истиной тогда и только тогда, когда `a` и `b` указывают на один и тот же объект.

Равенство структур

Структурное равенство проверяется оператором `==` (и его отрицанием `!=`). Условно, выражение `a == b` транслируется в:

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

Т.е. если `a` не `null`, вызывается функция `equals(Any?)`, иначе (т.е. если `a` указывает на `null`) `b`ссылочно сравнивается с `null`. Заметьте, что в явном сравнении с `null` для оптимизации нет смысла: `a == null` будет автоматически транслироваться в `a === null`.

Перегрузка операторов

Язык Kotlin позволяет нам реализовывать предопределённый набор операторов для наших типов. Эти операторы имеют фиксированное символическое представление (вроде `+` или `*`) и фиксированные приоритеты. Для реализации оператора мы предоставляем функцию-член или функцию-расширение с фиксированным именем и с соответствующим типом, т. е. левосторонним типом для бинарных операций или типом аргумента для унарных операций. Функции, которые перегружают операторы, должны быть отмечены модификатором `operator`.

Далее мы опишем соглашения, которые регламентируют перегрузку операторов для разных типов операторов.

Унарные операторы

Унарные префиксные операторы

Выражение	Транслируется в
`+a`	`a.unaryPlus()`
`-a`	`a.unaryMinus()`
`!a`	`a.not()`

Эта таблица демонстрирует, что когда компилятор обрабатывает, к примеру, выражение `+a`, он осуществляет следующие действия:

- Определяется тип выражения `a`, пусть это будет `T`.
- Смотрится функция `unaryPlus()` с модификатором `operator` без параметров для приёмника типа `T`, т. е. функция-член или функция расширения.
- Если функция отсутствует или неоднозначная, то это ошибка компиляции.

- Если функция присутствует и её возвращаемый тип есть `R`, выражение `+a` имеет Тип `R`.

Примечание: эти операции, как и все остальные, оптимизированы для основных типов и не вносят накладных расходов на вызовы этих функций для них.

Например, вы можете перегрузить оператор унарного минуса:

```
data class Point(val x: Int, val y: Int)


operator fun Point.unaryMinus() = Point(-x, -y)


val point = Point(10, 20)

println(-point)  // выведет "(-10, -20)"
```

Инкремент и декремент

Выражение	Транслируется в
`a++`	`a.inc()` + see below
`a--`	`a.dec()` + see below

Функции `inc()` и `dec()` должны возвращать значение, которое будет присвоено переменной, к которой была применёна операция `++` или `--` . Они не должны пытаться изменять сам объект, для которого `inc` или `dec` были вызваны.

Компилятор осуществляет следующие шаги для разрешения операторов в *постфиксной* форме, например для `a++`:

- Определяется тип переменной `a`, пусть это будет `T`.
- Смотрится функция `inc()` с модификатором `operator` без параметров, применимая для приёмника типа `T`.
- Проверяется, что возвращаемый тип такой функции является подтипом `T`.

Эффектом вычисления будет:

- Загружается инициализирующее значение `a` во временную переменную `a0`,
- Результат выполнения `a.inc()` сохраняется в `a`,
- Возвращается `a0` как результат вычисления выражения (т.е. значение до инкремента).

Для `a--` шаги выполнения полностью аналогичные.

Для *префиксной* формы `++a` или `--a` разрешение работает подобно, но результатом будет:

- Присвоение результата вычисления `a.inc()` непосредственно `a`,
- Возвращается новое значение `a` как общий результат вычисления выражения.

Бинарные операции

Арифметические операции

Выражение	Транслируется в
`a + b`	`a.plus(b)`
`a – b`	`a.minus(b)`
`a * b`	`a.times(b)`
`a / b`	`a.div(b)`
`a % b`	`a.rem(b)`, `a.mod(b)` (устаревшее)
`a..b`	`a.rangeTo(b)`

Для перечисленных в таблице операций компилятор всего лишь разрешает выражение из колонки *Транслируется в*.

Отметим, что операция `rem` поддерживается только начиная с Kotlin 1.1. Kotlin 1.0 использует только операцию `mod`, которая отмечена как устаревшая в Kotlin 1.1.

Пример

Ниже приведен пример класса Counter, начинающего счёт с заданного значения, которое может быть увеличено с помощью перегруженного оператора `+`.

```
data class Counter(val dayIndex: Int) {
    operator fun plus(increment: Int): Counter {
        return Counter(dayIndex + increment)
    }
}
```

Оператор in

Выражение	Транслируется в
`a in b`	`b.contains(a)`
`a !in b`	`!b.contains(a)`

Для операций `in` и `!in` используется одна и та же процедура, только возвращаемый результат инвертируется.

Оператор доступа по индексу

Выражение	Транслируется в
`a[i]`	`a.get(i)`
`a[i, j]`	`a.get(i, j)`
`a[i_1, ..., i_n]`	`a.get(i_1, ..., i_n)`
`a[i] = b`	`a.set(i, b)`
`a[i, j] = b`	`a.set(i, j, b)`
`a[i_1, ..., i_n] = b`	`a.set(i_1, ..., i_n, b)`

Квадратные скобки транслируются в вызов `get` или `set` с соответствующим числом аргументов.

Оператор вызова

Выражение	Транслируется в
`a()`	`a.invoke()`
`a(i)`	`a.invoke(i)`

Выражение	Транслируется в
`a(i, j)`	`a.invoke(i, j)`
`a(i_1, ..., i_n)`	`a.invoke(i_1, ..., i_n)`

Оператор вызова (функции, метода) в круглых скобках транслируется в `invoke` с соответствующим числом аргументов.

Присвоения с накоплением

Выражение	Транслируется в
`a += b`	`a.plusAssign(b)`
`a -= b`	`a.minusAssign(b)`
`a *= b`	`a.timesAssign(b)`
`a /= b`	`a.divAssign(b)`
`a %= b`	`a.modAssign(b)`

Для присваивающих операций, таких как `a += b`, компилятор осуществляет следующие шаги:

- Если функция из правой колонки таблицы доступна
  - Если соответствующая бинарная функция (т.е. `plus()` для `plusAssign()`) также доступна, то фиксируется ошибка (неоднозначность).
  - Проверяется, что возвращаемое значение функции `Unit`, в противном случае фиксируется ошибка.
  - Генерируется код для `a.plusAssign(b)`
- В противном случае делается попытка сгенерировать код для `a = a + b` (при этом включается проверка типов: тип выражения `a + b` должен быть подтипом `a`).

Отметим: присвоение *НЕ ЯВЛЯЕТСЯ* выражением в Kotlin.

Операторы равенства и неравенства

Выражение	Транслируется в
`a == b`	`a?.equals(b) ?: (b === null)`
`a != b`	`!(a?.equals(b) ?: (b === null))`

Отметим: операции `===` и `!==` (проверка идентичности) являются неперегружаемыми, поэтому не приводятся никакие соглашения для них.

Операция `==` имеет специальный смысл: она транслируется в составное выражение, в котором экранируются значения `null`. `null == null` - это всегда истина, а `x == null` для ненулевых `x` - всегда ложь, и не должно расширяться в `x.equals()`.

Операторы сравнений

Выражение	Транслируется в
`a > b`	`a.compareTo(b) > 0`
`a < b`	`a.compareTo(b) < 0`
`a >= b`	`a.compareTo(b) >= 0`
`a <= b`	`a.compareTo(b) <= 0`

Все сравнения транслируются в вызовы `compareTo`, от которых требуется возврат значения типа `Int`.

Инфиксные вызовы именованных функций

Мы можем моделировать вручную инфиксные операции использованием infix function calls.

Null безопасность

Nullable типы и Non-Null типы

Система типов в языке **Kotlin** нацелена на то, чтобы искоренить опасность обращения к *null* значениям, более известную как "Ошибка на миллион".

Самым распространённым подводным камнем многих языков программирования, в том числе **Java**, является попытка произвести доступ к *null* значению. Это приводит к ошибке. В **Java** такая ошибка называется `NullPointerException` (сокр. "NPE").

Kotlin призван исключить ошибки подобного рода из нашего кода. NPE могу возникать только в случае:

- Явного указания `throw NullPointerException()`
- Использования оператора `!!` (описано ниже)
- Эту ошибку вызвал внешний Java-код
- Есть какое-то несоответствие при инициализации данных (в конструкторе использована ссылка *this*на данные, которые не были ещё проинициализированы)

Система типов **Kotlin** различает ссылки на те, которые могут иметь значение *null* (nullable ссылки) и те, которые таковыми быть не могут (non-null ссылки). К примеру, переменная часто используемого типа `String` не может быть *null*:

```
var a: String = "abc"

a = null // ошибка компиляции
```

Для того, чтобы разрешить *null* значение, мы можем объявить эту строковую переменную как `String?`:

```
var b: String? = "abc"

b = null // ok
```

Теперь, при вызове метода с использованием переменной `a`, исключены какие-либо NPE. Вы спокойно можете писать:

```
val l = a.length
```

Но в случае, если вы захотите получить доступ к значению `b`, это будет небезопасно. Компилятор предупредит об ошибке:

```
val l = b.length // ошибка: переменная `b` может быть null
```

Но нам по-прежнему надо получить доступ к этому свойству/значению, так? Есть несколько способов этого достичь.

Проверка на *null*

Первый способ. Вы можете явно проверить `b` на *null* значение и обработать два варианта по отдельности:

```
val l = if (b != null) b.length else -1
```

Компилятор отслеживает информацию о проведённой вами проверке и позволяет вызывать `lenght`внутри блока *if*. Также поддерживаются более сложные конструкции:

```
if (b != null && b.length > 0) {
    print("String of length ${b.length}")
} else {
    print("Empty string")
}
```

Обратите внимание: это работает только в том случае, если `b` является неизменной переменной (ориг.: *immutable*). Например, если это локальная переменная, значение которой не изменяется в период между его проверкой и использованием. Также такой переменной может служить *val*. В противном случае может так оказаться, что переменная `b` изменила своё значение на *null* после проверки.

Безопасные вызовы

Вторым способом является оператор безопасного вызова `?.`:

`kotlin b?.length`

Этот код возвращает `b.lenght` в том, случае, если `b` не имеет значение *null*. Иначе он возвращает *null*. Типом этого выражения будет `Int?`.

Такие безопасные вызовы полезны в цепочках. К примеру, если Bob, Employee (работник), может быть прикреплён (или нет) к отделу Department, и у отдела может быть управляющий, другой Employee. Для того, чтобы обратиться к имени этого управляющего (если такой есть), напишем:

```
bob?.department?.head?.name
```

Такая цепочка вернёт *null* в случае, если одно из свойств имеет значение *null*.

Для проведения каких-либо операций над non-null значениями вы можете использовать `let` оператор вместе с оператором безопасного вызова:

```
val listWithNulls: List<String?> = listOf("A", null)
for (item in listWithNulls) {
    item?.let { println(it) } // выводит A и игнорирует null
}
```

Элвис-оператор

Если у нас есть nullable ссылка `r`, мы можем либо провести проверку этой ссылки и использовать её, либо использовать non-null значение `x`:

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

Аналогом такому *if*-выражению является элвис-оператор `?:`:

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

Если выражение, стоящее слева от Элвис-оператора, не является null, то элвис-оператор его вернёт. В противном случае, в качестве возвращаемого значения послужит то, что стоит справа. Обращаем ваше внимание на то, что часть кода, расположенная справа, выполняется ТОЛЬКО в случае, если слева получается null.

Так как *throw* и *return* тоже являются выражениями в **Kotlin**, их также можно использовать справа от Элвис-оператора. Это может быть крайне полезным для проверки аргументов функции:

```
fun foo(node: Node): String? {

    val parent = node.getParent() ?: return null

    val name = node.getName() ?: throw IllegalArgumentException("name exp
ected")

    // ...

}
```

Оператор `!!`

Для любителей NPE сушествует ещё один способ. Мы можем написать `b!!` и это вернёт нам либо non-null значение `b` (в нашем примере вернётся `String`), либо выкинет NPE:

```
val l = b!!.length
```

В случае, если вам нужен NPE, вы можете заполучить её только путём явного указания.

Безопасные приведения типов (ориг.: *Safe Casts*)

Обычное приведение типа может вызвать `ClassCastException` в случае, если объект имеет другой тип. Можно использовать безопасное приведение, которое вернёт *null*, если попытка не удалась:

```
val aInt: Int? = a as? Int
```

Коллекции nullable типов

Если у вас есть коллекция nullable элементов и вы хотите отфильтровать все non-null элементы, используйте фунуцию `filterNotNull`.

```
val nullableList: List<Int?> = listOf(1, 2, null, 4)
val intList: List<Int> = nullableList.filterNotNull()
```

Исключения

Классы исключений

Все исключения в **Kotlin** являются наследниками класса `Throwable`. У каждого исключения есть сообщение, трассировка стека, а также причина, по которой это исключение вероятно было вызвано.

Для того, чтобы возбудить исключение явным образом, используйте оператор *throw*

```
throw MyException("Hi There!")
```

Оператор *try* позволяет перехватывать исключения

```
try {
    // some code
}
catch (e: SomeException) {
    // handler
}
finally {
    // optional finally block
}
```

В коде может быть любое количество блоков *catch* (такие блоки могут и вовсе отсутствовать). Блоки *finally*могут быть опущены. Однако, должен быть использован как минимум один блок *catch* или *finally*.

Try - это выражение

Ключевое слово *try* является выражением, то есть оно может иметь возвращаемое значение.

```
val a: Int? = try { parseInt(input) } catch (e: NumberFormatException) { null }
```

Возвращаемым значением будет либо последнее выражение в блоке *try*, либо последнее выражение в блоке *catch* (или блоках). Содержимое *finally* блока никак не повлияет на результат *try*-выражения.

Проверяемые исключения

В языке **Kotlin** нет проверяемых исключений. Для этого существует целый ряд причин, но мы рассмотрим простой пример.

Приведённый ниже фрагмент кода является примером простого интерфейса в JDK, который реализован в классе `StringBulder`

```
Appendable append(CharSequence csq) throws IOException;
```

О чём говорит нам сигнатура? О том, что каждый раз, когда я присоединяю строку к чему-то (к `StringBuilder`, какому-нибудь логу, сообщению в консоль и т.п) , мне необходимо отлавливать исключения типа `IOExceptions`. Почему? Потому, что данная операция может вызывать IO (Input-Output: Ввод-Вывод) (`Writer` также реализует интерфейс `Appendable`)... Данный факт постоянно приводит к написанию подобного кода:

```
try {
    log.append(message)
}
catch (IOException e) {
    // Должно быть безопасно
}
```

И это плохо. См. Effective Java, Item 65: *Don't ignore exceptions* (не игнорируйте исключения).

Брюс Эккель как-то сказал в своей статье "Нужны ли в Java проверяемые исключения?"(Does Java need Checked Exceptions?):

> Анализ небольших программ показал, что обязательная обработка исключений может повысить производительность разработчика и улучшить качество кода. Однако, изучение крупных проектов по разработке программного обеспечения позволяет сделать противоположный вывод: происходит понижение продуктивности и сравнительно небольшое улучшение кода (а иногда и без всякого улучшения).

Другие цитаты подобного рода:

- Java's checked exceptions were a mistake (Rod Waldhoff)
- The Trouble with Checked Exceptions (Anders Hejlsberg)

Тип `Nothing`

Вы можете использовать выражение `throw` в качестве части элвис-выражения:

```
val s = person.name ?: throw IllegalArgumentException("Name required")
```

Типом выражения `throw` является специальный тип под названием `Nothing`. У этого типа нет никаких значений, он используетя для того, чтобы обозначить те участки кода, которые могут быть не достигнуты никогда. В своём коде вы можете использовать `Nothing` для того, чтобы отметить функцию, чей результат никогда не будет возвращён:

```
fun fail(message: String): Nothing {

    throw IllegalArgumentException(message)

}
```

При вызове такой функции компилятор будет в курсе, что исполнения кода далее не последует:

```
val s = person.name ?: fail("Name required")

println(s)     // известно, что переменная 's' проинициализирована к этому моменту
```

Совместимость с **Java**

См. раздел, посвещённый исключениям, в "Совместимости с Java" Java Interoperability section.

Аннотации

Объявление аннотаций

Аннотации являются специальной формой синтаксических метаданных, добавленных в исходный код. Для объявления аннотации используйте модификатор *annotation* перед именем класса:

```
annotation class Fancy
```

Дополнительные атрибуты аннотаций могут быть определены путём аннотации класса-аннотации мета-аннотациями:

- `@Target` определяет возможные виды элементов, которые могут быть помечены аннотацией (классы, функции, свойства, выражения и т.д.);
- `@Retention` определяет, будет ли аннотация храниться в скомпилированном классе и будет ли видима через рефлексию (по умолчанию оба утверждения верны);
- `@Repeatable` позволяет использовать одну и ту же аннотацию на одном элементе несколько раз;
- `@MustBeDocumented` определяет то, что аннотация является частью публичного API и должна быть включена в сигнатуру класса или метода, попадающую в сгенерированную документацию.

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
```

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

Использование

```
@Fancy class Foo {
    @Fancy fun baz(@Fancy foo: Int): Int {
        return (@Fancy 1)
    }
}
```

Если вам нужно пометить аннотацией первичный конструктор класса, следует добавить ключевое слово *constructor* при объявлении конструктора и вставить аннотацию перед ним:

```
class Foo @Inject constructor(dependency: MyDependency) {
    // ...
}
```

Вы также можете помечать аннотациями геттеры и сеттеры:

```
class Foo {
    var x: MyDependency? = null
        @Inject set
}
```

Конструкторы

Аннотации могут иметь конструкторы, принимающие параметры:

```
annotation class Special(val why: String)


@Special("пример") class Foo {}
```

Разрешены параметры следующих типов:

- типы, которые соответствуют примитивам Java (Int, Long, и т.д.);
- строки;
- классы (`Foo::class`);
- перечисляемые типы;
- другие аннотации;

- массивы, содержащие значения приведённых выше типов.

Параметры аннотаций не могут иметь обнуляемые типы, потому что JVM не поддерживает хранение `null` в качестве значения атрибута аннотации.

Если аннотация используется в качестве параметра к другой аннотации, её имя не нужно начинать со знака @:

```
annotation class ReplaceWith(val expression: String)


annotation class Deprecated(
        val message: String,
        val replaceWith: ReplaceWith = ReplaceWith(""))


@Deprecated("Эта функция устарела, вместо неё используйте ===", ReplaceWit
h("this === other"))
```

Если вам нужно определить класс как аргумент аннотации, используйте Kotlin класс (KClass). Компилятор Kotin автоматически сконвертирует его в Java класс, так что код на Java сможет видеть аннотации и их аргументы.

```
import kotlin.reflect.KClass


annotation class Ann(val arg1: KClass<*>, val arg2: KClass<out Any?>)


@Ann(String::class, Int::class) class MyClass
```

Лямбды

Аннотации также можно использовать с лямбдами. Они будут применены к `invoke()` методу, в который генерируется тело лямбды. Это полезно для фреймворков вроде Quasar, который использует аннотации для контроля многопоточности.

```
annotation class Suspendable


val f = @Suspendable { Fiber.sleep(10) }
```

Аннотации с указаниями

Когда вы помечаете свойство или первичный конструктор аннотацией, из соответствующего Kotlin-элемента генерируются несколько Java-элементов, и поэтому в сгенерированном байт-коде элемент появляется в нескольких местах. Чтобы указать, в

каком именно месте аннотация должна быть сгенерирована, используйте следующий синтаксис:

```
class Example(@field:Ann val foo,    // аннотация для Java-поля
              @get:Ann val bar,      // аннотация для Java-геттера
              @param:Ann val quux)   // аннотация для параметра конструкто
ра Java
```

Тот же синтаксис может быть использован для аннотации целого файла. Для этого отметьте аннотацию словом `file` и вставьте её в начале файла: перед указанием пакета или перед импортами, если файл находится в пакете по умолчанию:

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


package org.jetbrains.demo
```

Если вы помечаете аннотацией несколько элементов, вы можете избежать повторения путём указания всех аннотаций в квадратных скобках:

```
class Example {
    @set:[Inject VisibleForTesting]
    var collaborator: Collaborator
}
```

Полный список поддерживаемых указаний:

- `file`
- `property` (такие аннотации не будут видны в Java)
- `field`
- `get` (геттер)
- `set` (сеттер)
- `receiver` (параметр-приёмник расширения)
- `param` (параметр конструктора)
- `setparam` (параметр сеттера)
- `delegate` (поле, которое хранит экземпляр делегата для делегированного свойства)

Чтобы пометить аннотацией параметр-приёмник расширения, используйте следующий синтаксис:

```
fun @receiver:Fancy String.myExtension() { }
```

Если вы не сделали указание, аннотация будет применена к элементу, выбранному в соответствии с аннотацией `@Target` той аннотации, которую вы используете. Если существует несколько элементов, к которым возможно применение аннотации, будет выбран первый подходящий элемент из следующего списка:

- `param`
- `property`

- `field`

Java-аннотации

Java-аннотации на 100% совместимы в Kotlin:

```
import org.junit.Test
import org.junit.Assert.*
import org.junit.Rule
import org.junit.rules.*

class Tests {
    // применение аннотации @Rule к геттеру
    @get:Rule val tempFolder = TemporaryFolder()

    @Test fun simple() {
        val f = tempFolder.newFile()
        assertEquals(42, getTheAnswer())
    }
}
```

Так как порядок параметров для Java-аннотаций не задан, вы не можете использовать обычный синтаксис вызова функции для передачи аргументов. Вместо этого вам нужно использовать именованные аргументы.

```
// Java
public @interface Ann {
    int intValue();
    String stringValue();
}
// Kotlin
@Ann(intValue = 1, stringValue = "abc") class C
```

Также, как и в Java, параметр `value` — особый случай; его значение может быть определено без явного указания имени.

```
// Java
public @interface AnnWithValue {
```

```
    String value();
}
// Kotlin
@AnnWithValue("abc") class C
```

Если аргумент `value` в Java является массивом, в Kotlin он становится `vararg` параметром:

```
// Java
public @interface AnnWithArrayValue {
    String[] value();
}
// Kotlin
@AnnWithArrayValue("abc", "foo", "bar") class C
```

Для прочих аргументов, которые являются массивом, вам нужно явно использовать `arrayOf`:

```
// Java
public @interface AnnWithArrayMethod {
    String[] names();
}
// Kotlin
@AnnWithArrayMethod(names = arrayOf("abc", "foo", "bar")) class C
```

Значения экземпляра аннотации становятся свойствами в Kotlin-коде.

```
// Java
public @interface Ann {
    int value();
}
// Kotlin
fun foo(ann: Ann) {
    val i = ann.value
}
```

Рефлексия

Рефлексия — это набор возможностей языка и библиотек, который позволяет интроспектировать программу (обращаться к её структуре) во время её исполнения. В

Kotlin функции и свойства первичны, и поэтому их интроспекция (например, получение имени или типа во время исполнения) сильно переплетена с использованием функциональной или реактивной парадигмы.

На платформе Java библиотека для использования рефлексии находится в отдельном JAR-файле (`kotlin-reflect.jar`). Это было сделано для уменьшения требуемого размера runtime-библиотеки для приложений, которые не используют рефлексию. Если вы используете рефлексию, удостоверьтесь, что этот .jar файл добавлен в classpath вашего проекта.

Ссылки на классы

Самая базовая возможность рефлексии — это получение ссылки на Kotlin класс. Чтобы получить ссылку на статический Kotlin класс, используйте синтаксис *литерала класса*:

```
val c = MyClass::class
```

Ссылка на класс имеет тип KClass.

Обратите внимание, что ссылка на Kotlin класс это не то же самое, что и ссылка на Java класс. Для получения ссылки на Java класс, используйте свойство `.java` экземпляра `KClass`.

Ссылки на привязанные классы

Примечание: доступно с версии Kotlin 1.1

Вы можете получить ссылку на класс определённого объекта с помощью уже известного вам синтаксиса, вызвав `::class` у нужного объекта:

```
val widget: Widget = ...

assert(widget is GoodWidget) { "Bad widget: ${widget::class.qualifiedName}
" }
```

Вы получите ссылку на точный класс объекта, например `GoodWidget` или `BadWidget`, несмотря на тип объекта, участвующего в выражении (`Widget`).

Ссылки на функции

Когда у нас есть именованная функция, объявленная следующим образом:

```
fun isOdd(x: Int) = x % 2 != 0
```

Мы можем как вызвать её напрямую (`isOdd(5)`), так и передать её как значение, например в другую функцию. Чтобы сделать это, используйте оператор `::`:

```
val numbers = listOf(1, 2, 3)

println(numbers.filter(::isOdd)) // выведет [1, 3]
```

Здесь, `::isOdd` — значение функционального типа `(Int) -> Boolean`.

Оператор `::` может быть использован с перегруженными функциями, когда тип используемой функции известен из контекста. Например:

```
fun isOdd(x: Int) = x % 2 != 0

fun isOdd(s: String) = s == "brillig" || s == "slithy" || s == "tove"


val numbers = listOf(1, 2, 3)

println(numbers.filter(::isOdd)) // ссылается на isOdd(x: Int)
```

Также вместо этого вы можете указать нужный контекст путём сохранения ссылки на функцию в переменной, тип которой задан явно:

```
val predicate: (String) -> Boolean = ::isOdd   // ссылается на isOdd(x: St
ring)
```

Если вы хотите использовать член класса или функцию-расширение, вам нужно обозначить это явным образом. Например, `String::toCharArray` даёт нам функцию-расширение для типа `String`: `String.() -> CharArray`

Пример: композиция функций

Рассмотрим следующую функцию:

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {

    return { x -> f(g(x)) }

}
```

Она возвращает композицию двух функций, переданных ей: `compose(f, g) = f(g(*))`. Теперь вы можете применять её к ссылкам на функции:

```
fun length(s: String) = s.length


val oddLength = compose(::isOdd, ::length)

val strings = listOf("a", "ab", "abc")


println(strings.filter(oddLength)) // выведет "[a, abc]"
```

Ссылки на свойства

Для доступа к свойствам как первичным объектам в Kotlin мы по-прежнему можем использовать оператор `::`:

```
var x = 1
```

```
fun main(args: Array<String>) {
    println(::x.get()) // выведет "1"
    ::x.set(2)
    println(x)         // выведет "2"
}
```

Выражение `::x` возвращает объект свойства типа `KProperty<Int>`, который позволяет нам читать его значение с помощью `get()` или получать имя свойства при помощи обращения к свойству `name`. Для получения более подробной информации обратитесь к документации класса `KProperty`.

Для изменяемых свойств, например `var y = 1`, `::y` возвращает значение типа`KMutableProperty<Int>`.

Ссылка на свойство может быть использована там, где ожидается функция без параметров:

```
val strs = listOf("a", "bc", "def")
println(strs.map(String::length)) // выведет [1, 2, 3]
```

Для доступа к свойству, которое является членом класса, мы указываем класс:

```
class A(val p: Int)

fun main(args: Array<String>) {
    val prop = A::p
    println(prop.get(A(1))) // выведет "1"
}
```

Для функции-расширения:

```
val String.lastChar: Char
    get() = this[length - 1]

fun main(args: Array<String>) {
    println(String::lastChar.get("abc")) // выведет "c"
}
```

Взаимодействие с рефлексией Java

На платформе Java стандартная библиотека Kotlin содержит расширения, которые сопоставляют расширяемые ими объекты рефлексии Kotlin с объектами рефлексии Java

(см. пакет `kotlin.reflect.jvm`). К примеру, для нахождения поля или метода, который служит геттером для Kotlin-свойства, вы можете написать что-то вроде этого:

```
import kotlin.reflect.jvm.*


class A(val p: Int)


fun main(args: Array<String>) {
    println(A::p.javaGetter) // выведет "public final int A.getP()"
    println(A::p.javaField)  // выведет "private final int A.p"
}
```

Для получения класса Kotlin, соответствующего классу Java, используйте свойство-расширение `.kotlin`:

```
fun getKClass(o: Any): KClass<Any> = o.javaClass.kotlin
```

Ссылки на конструктор

К конструкторам можно обратиться так же, как и к методам или свойствам. Они могут быть использованы везде, где ожидается объект функционального типа. Обращение к конструкторам происходит с помощью оператора `::` и имени класса. Рассмотрим функцию, которая принимает функциональный параметр без параметров и возвращает `Foo`:

```
class Foo


fun function(factory : () -> Foo) {
    val x : Foo = factory()
}
```

Используя `::Foo`, конструктор класса Foo без аргументов, мы можем просто вызывать функцию таким образом:

```
function(::Foo)
```

Привязанные функции

Вы можете сослаться на экземпляр метода конкретного объекта.

```
val numberRegex = "\\d+".toRegex()
println(numberRegex.matches("29")) // выведет "true"
```

```
val isNumber = numberRegex::matches

println(isNumber("29")) // выведет "true"
```

Вместо вызова метода `matches` напрямую, мы храним ссылку на него. Такие ссылки привязаны к объектам, к которым относятся:

```
val strings = listOf("abc", "124", "a70")

println(strings.filter(numberRegex::matches)) // выведет "[124]"
```

Сравним типы привязанных и соответствующих непривязанных ссылок. Объект-приёмник "прикреплён" к привязанной ссылке, поэтому тип приёмника больше не является параметром:

```
val isNumber: (CharSequence) -> Boolean = numberRegex::matches


val matches: (Regex, CharSequence) -> Boolean = Regex::matches
```

Ссылка на свойство может быть также привязанной:

```
val prop = "abc"::length

println(prop.get())   // выведет "3"
```

Типобезопасные строители

Идея строителей (builders) довольна популярна в сообществе *Groovy*. Строители позволяют объявлять данные в полудекларативном виде. Строители хороши для генерации XML, вёрстки компонентов UI, описания 3D сцен и многого другого...

В отличие от Groovy, Kotlin проверяет типы строителей, что делает их более приятными в использовании в большинстве юзкейсов.

Для прочих случаев Kotlin поддерживает Динамически типизированные строители (Dynamic types builders).

Пример типобезопасного строителя

Рассмотрим следующий код:

```
import com.example.html.* // смотрите объявления ниже


fun result(args: Array<String>) =

    html {

        head {

            title {+"XML кодирование с Kotlin"}
```

```
        }
        body {
            h1 {+"XML кодирование с Kotlin"}
            p  {+"этот формат может быть использован как альтернатва XML"}

            // элемент с атрибутом и текстовым содержанием
            a(href = "http://kotlinlang.ru") {+"Kotlin"}

            // смешанный контент
            p {
                +"Немного"
                b {+"смешанного"}
                +"текста. Посмотрите наш"
                a(href = "http://kotlinlang.org") {+"перевод"}
                +"документации Kotlin."
            }
            p {+"немного текста"}

            // контент генерируется в цикле
            p {
                for (arg in args)
                    +arg
            }
        }
    }
```

Всё это полностью корректный Kotlin-код. Здесь вы можете отредактировать и запустить пример с этим кодом прямо у себя в браузере.

Как это работает

Давайте рассмотрим механизм реализации типобезопасных строителей в Kotlin. Прежде всего, нам нужно определить модель, которую мы собираемся строить. В данном случае это HTML-тэги. Мы можем сделать это без труда с помощью нескольких классов. К

примеру, `HTML` — это класс, который описывает тэг `<html>`, т.е. он определяет потомков, таких как `<head>` и `<body>`. (См. его объявление ниже.)

Теперь давайте вернёмся к вопросу почему мы можем писать вот такой код:

```
html {
 // ...
}
```

На самом деле, `html` является вызовом функции, которая принимает лямбда-выражение в качестве аргумента. Вот как эта функция определена:

```
fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

Эта функция принимает один параметр-функцию под названием `init`. Тип этой функции: `HTML.() -> Unit` — *функциональный тип с объектом-приёмником*. Это значит, что нам нужно передать экземпляр класса `HTML` (*приёмник*) в функцию, и мы сможем обращаться к членам объекта в теле этой функции. Обращение происходит через ключевое слово **this**:

```
html {
    this.head { /* ... */ }
    this.body { /* ... */ }
}
```

(`head` и `body` — члены класса `HTML`)

Теперь **this** может быть опущено, и мы получим что-то, что уже очень похоже на строителя:

```
html {
    head { /* ... */ }
    body { /* ... */ }
}
```

Итак, что же делает этот вызов? Давайте посмотрим на тело функции `html`, объявленной выше. Она создаёт новый экземпляр `HTML`, затем инициализирует его путём вызова функции, которая была передана в аргументе (в нашем примере это сводится к вызову `head` и `body` у объекта `HTML`), и после этого возвращает его значение. Это в точности то, что и должен делать строитель.

Функции `head` и `body` в классе `HTML` объявлены схоже с функцией `html`. Единственное отличие в том, что они добавляют отстроенные экземпляры в коллекцию `children` заключающего экземпляра `HTML`:

```
fun head(init: Head.() -> Unit) : Head {
    val head = Head()
    head.init()
    children.add(head)
    return head
}

fun body(init: Body.() -> Unit) : Body {
    val body = Body()
    body.init()
    children.add(body)
    return body
}
```

На самом деле эти две функции делают одно и тоже, поэтому мы можем использовать обобщённую версию, `initTag`:

```
protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
    tag.init()
    children.add(tag)
    return tag
}
```

Теперь наши функции выглядят очень просто:

```
fun head(init: Head.() -> Unit) = initTag(Head(), init)

fun body(init: Body.() -> Unit) = initTag(Body(), init)
```

И мы можем использовать их для постройки тэгов `<html>` и `<body>`.

Ещё одна вещь, которую следует обсудить, это добавление текста в тело тэга. В примере выше мы используем такой синтаксис:

```
html {
    head {
```

```
        title {+"XML кодирование с Kotlin"}
    }
    // ...
}
```

Итак, мы просто добавляем строку в тело тэга, приписав `+` перед текстом, что ведёт к вызову префиксной операции `unaryPlus()`. Эта операция определена с помощью функции-расширения `unaryPlus()`, которая является членом абстрактного класса `TagWithText` (родителя `Title`).

```
fun String.unaryPlus() {
    children.add(TextElement(this))
}
```

Иными словами, префикс `+` оборачивает строку в экземпляр `TextElement` и добавляет его в коллекцию `children`.

Всё это определено в пакете `com.example.html`, который импортирован в начале примера выше. В последней секции вы можете прочитать полное описание определений в этом пакете.

Контроль области видимости: @DslMarker (с версии 1.1)

При использовании DSL может возникнуть проблема, когда слишком много функций могут быть вызваны в определённом контексте. Мы можем вызывать методы каждого неявного приёмника внутри лямбды, и из-за этого может возникать противоречивый результат, как, например, тэг `head` внутри другого тэга `head`:

```
html {
    head {
        head {} // должен быть запрещён
    }
    // ...
}
```

В этом примере должны быть доступны только члены ближайшего неявного приёмника `this@head`; `head()` является членом другого приёмника — `this@html`, поэтому его вызов в другом контексте должен быть запрещён.

Для решения этой проблемы в Kotlin 1.1 был введен специальный механизм для управления областью приёмника.

Чтобы заставить компилятор запускать контрольные области, нам нужно только аннотировать типы всех получателей, используемых в DSL, той же маркерной аннотацией. Например, для HTML Builders мы объявляем аннотацию `@HTMLTagMarker`:

```
@DslMarker

annotation class HtmlTagMarker
```

Аннотированный класс называется DSL-маркером, если он помечен аннотацией `@DslMarker`.

В нашем DSL все классы тэгов расширяют один и тот же суперкласс `Tag`. Нам достаточно аннотировать `@HtmlTagMarker` только суперкласс, и после этого компилятор Kotlin обработает все унаследованные классы в соответствии с аннотацией:

```
@HtmlTagMarker

abstract class Tag(val name: String) { ... }
```

Нам не нужно помечать классы `HTML` или `Head` аннотацией `@HtmlTagMarker`, потому что их суперкласс уже аннотирован:

```
class HTML() : Tag("html") { ... }

class Head() : Tag("head") { ... }
```

После добавления этой аннотации, компилятор Kotlin знает, какие неявные приёмники являются частью того же DSL, и разрешает обращаться только к членам ближайших приёмников:

```
html {

    head {

        head { } // ошибка: член внешнего приёмника

    }

    // ...

}
```

Обратите внимание, что всё ещё возможно вызывать члены внешнего приёмника, но для этого вам нужно указать этот приёмник явно:

```
html {

    head {

        this@html.head { } // всё работает

    }

    // ...

}
```

Полное описание пакета `com.example.html`

Перед вами содержание пакета `com.example.html` (представлены только элементы, использованные в примере выше). Он строит HTML дерево и активно использует расширения и лямбды с приёмниками.

Примечание: Аннотация @DslMarker доступна в Kotlin начиная с версии 1.1

```
package com.example.html

interface Element {
    fun render(builder: StringBuilder, indent: String)
}

class TextElement(val text: String) : Element {
    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent$text\n")
    }
}

@DslMarker
annotation class HtmlTagMarker

@HtmlTagMarker
abstract class Tag(val name: String) : Element {
    val children = arrayListOf<Element>()
    val attributes = hashMapOf<String, String>()

    protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
        tag.init()
        children.add(tag)
        return tag
    }
```

```
    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent<$name${renderAttributes()}>\n")
        for (c in children) {
            c.render(builder, indent + "  ")
        }
        builder.append("$indent</$name>\n")
    }

    private fun renderAttributes(): String {
        val builder = StringBuilder()
        for ((attr, value) in attributes) {
            builder.append(" $attr=\"$value\"")
        }
        return builder.toString()
    }

    override fun toString(): String {
        val builder = StringBuilder()
        render(builder, "")
        return builder.toString()
    }
}

abstract class TagWithText(name: String) : Tag(name) {
    operator fun String.unaryPlus() {
        children.add(TextElement(this))
    }
}

class HTML : TagWithText("html") {
```

```
    fun head(init: Head.() -> Unit) = initTag(Head(), init)

    fun body(init: Body.() -> Unit) = initTag(Body(), init)
}

class Head : TagWithText("head") {
    fun title(init: Title.() -> Unit) = initTag(Title(), init)
}

class Title : TagWithText("title")

abstract class BodyTag(name: String) : TagWithText(name) {
    fun b(init: B.() -> Unit) = initTag(B(), init)
    fun p(init: P.() -> Unit) = initTag(P(), init)
    fun h1(init: H1.() -> Unit) = initTag(H1(), init)
    fun a(href: String, init: A.() -> Unit) {
        val a = initTag(A(), init)
        a.href = href
    }
}

class Body : BodyTag("body")
class B : BodyTag("b")
class P : BodyTag("p")
class H1 : BodyTag("h1")

class A : BodyTag("a") {
    var href: String
        get() = attributes["href"]!!
        set(value) {
            attributes["href"] = value
```

```
        }
}


fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

Псевдонимы типов

Примечание: Псевдонимы типов доступны в Kotlin начиная с версии 1.1

Псевдонимы типов предоставляют альтернативные имена для существующих типов. Если имя типа слишком длинное, вы можете ввести другое, более короткое имя, и использовать его вместо первоначального.

Псевдонимы типов полезны, когда вы хотите сократить длинные имена типов, содержащих обобщения. К примеру, можно упрощать названия типов коллекций:

```
typealias NodeSet = Set<Network.Node>


typealias FileTable<K> = MutableMap<K, MutableList<File>>
```

Вы также можете объявить псевдонимы для функциональных типов:

```
typealias MyHandler = (Int, String, Any) -> Unit


typealias Predicate<T> = (T) -> Boolean
```

Вы можете ввести новые имена для внутренних или вложенных классов:

```
class A {
    inner class Inner
}
class B {
    inner class Inner
}


typealias AInner = A.Inner
```

```
typealias BInner = B.Inner
```

Псевдонимы типов не вводят новых типов. Они эквивалентны соответствующим базовым типам. Когда вы добавляете `typealias Predicate<T>` и используете `Predicate<Int>` в своём коде, компилятор Kotlin всегда преобразовывает это в `(Int) -> Boolean`. Таким образом, вы можете передать переменную своего типа в любое место, где требуется базовый тип (тот, которому был задан псевдоним), и наоборот:

```
typealias Predicate<T> = (T) -> Boolean


fun foo(p: Predicate<Int>) = p(42)


fun main(args: Array<String>) {
    val f: (Int) -> Boolean = { it > 0 }
    println(foo(f)) // выведет "true"

    val p: Predicate<Int> = { it > 0 }
    println(listOf(1, -2).filter(p)) // выведет "[1]"
}
```

Сравнение с языком программирования Java

Некоторые проблемы Java, решённые в Kotlin

Koltin решает целый ряд проблем, от которых страдает Java:

- Ссылки на null контролируются системой типов.
- Нет сырых (raw) типов
- Массивы в Kotlin инвариантны
- Kotlin имеет правильные функциональные типы и поддерживает их использование вместо SAM-типов из Java
- Вариативность на месте использования без подстановочных символов (или масок, ориг.: *wildcards*)
- В Kotlin нет проверяемых исключений

Что есть в Java, но нет в Kotlin

- Проверяемые исключения
- Примитивные типы, которые не являются классами
- Статичные члены
- Не-приватные поля
- Подстановочные символы (маски, wildcards)

Что есть в Kotlin, но нет в Java

- Лямбда-выражения + inline-функции = производительные и контролируемые пользовательские структуры
- Функции-расширения
- Null-безопасность
- Умные приведения
- Строковые шаблоны
- Свойства
- Первичный конструктор
- Делегирование на уровне языка
- Выведение типа для переменных и свойств
- Синглтоны на уровне языка
- Вариативность на уровне объявления и Проекции типов
- Интервалы
- Перегрузка операторов
- Вспомогательные объекты
- Классы данных
- Раздельные интерфейсы для изменяемых и неизменяемых коллекций
- Сопрограммы (корутины)

Редактирование статей

Редактировать статьи можно на GitHub. Например, чтобы отредактировать *эту* страницу откройте ссылку https://github.com/phplego/kotlinlang.ru/edit/master/how-to-edit.md или нажмите "Редактировать на GitHub" справа вверху.

После того, как вы отредактировали статью на GitHub, её нужно выгрузить на сервер kotlinlang.ru. Для этого перейдите по ссылке http://kotlinlang.ru/update.php

Почему некоторые статьи еще на английском?

Потому что их еще никто не перевел :(. Начните их переводить, это действительно нужно делать. Также нужно улучшать существующий перевод. Спасибо.

Создание новой статьи

Для создания новой статьи нужно выполнить два действия:

- Добавить пункт меню в файл `menu.json`. То есть отредактировать `menu.json`https://github.com/phplego/kotlin.su/edit/master/menu.json
- Создать файл с соответствующим именем `<article-id>.md`. Или скопировать соответствующий файл из английской документации.

Оригинальные тексты документации на английском находятся тут: https://github.com/JetBrains/kotlin-web-site/tree/master/pages/docs/reference

Ну и конечно же после любых изменений нужно:

- Сделать фиксацию изменений в git
- Если вы работаете с локальной копией, то сделать push
- Выгрузить файлы на сервер kotlinlang.ru перейдя по урлу: http://kotlinlang.ru/update.php

У меня нет доступа на редактирование, что делать?

В этом случае есть два варианта:

- Попросить доступ в чате https://telegram.me/KotlinLangRu
- Сделать Pull Request ваших изменений, и мы их одобрим (или отклоним) в ближайшее время

.

Участники

Oleg Dubrov

Krasnodar

Земляков Станислав / Zemlyakov Stanislav

Russian Federation, Moscow

Mikhail Malyshev

Moscow, Russia

Artem

Russia

Sergei Tsypanov

Olej

Ukraine, Kharkov

Grigory Bondarenko

Alexey Pyltsyn

Rostov-on-Don, Russia

Andrew Black

Ukraine

Andrey Netyaga

Saint Petersburg

Alex Ilyenko

Kiev, Ukraine

MaxF

Russia

Петров Александр

Novosibirsk

Последние изменения

05 июля 2017 г., 13:09:50 phplego: Merge pull request #47 from FilenkovMaxim/patch-1 Update basic-syntax.md 9cd5660

05 июля 2017 г., 11:43:02 FilenkovMaxim: Update basic-syntax.md опечатка a44eba1

25 июня 2017 г., 20:37:04 phplego: string anchor added ad5a1d2

25 июня 2017 г., 20:27:38 phplego: Rename menu.json to MENU.json 9f10952

23 июня 2017 г., 23:27:04 phplego: Update INDEX.md добавлено - Перевести статьи из радела "Java Interop" 7fa1375

20 июня 2017 г., 20:55:28 Temon137: Update INDEX.md Были проверены и исправлены все якоря. d4c14a8

20 июня 2017 г., 20:54:58 Temon137: Update coroutines.md 4b5416d

20 июня 2017 г., 20:43:06 Temon137: Update generics.md 9cf8951

20 июня 2017 г., 20:39:36 Temon137: Update visibility-modifiers.md fa4b352

20 июня 2017 г., 20:39:22 Temon137: Update typecasts.md 3008eb3

20 июня 2017 г., 20:39:14 Temon137: Update returns.md 14914b2

20 июня 2017 г., 20:39:06 Temon137: Update properties.md f580b87

20 июня 2017 г., 20:38:54 Temon137: Update packages.md 18fe54a

20 июня 2017 г., 20:38:46 Temon137: Update nested-classes.md 551130c

20 июня 2017 г., 20:38:34 Temon137: Update lambdas.md 3d8ec79

20 июня 2017 г., 20:38:26 Temon137: Update inline-functions.md cbd585f

20 июня 2017 г., 20:38:05 Temon137: Update extensions.md bccf5f1

20 июня 2017 г., 20:37:53 Temon137: Update delegated-properties.md 840def2

20 июня 2017 г., 20:37:35 Temon137: Update data-classes.md 624a422

20 июня 2017 г., 20:37:23 Temon137: Update coroutines.md 8426154

20 июня 2017 г., 20:37:08 Temon137: Update control-flow.md 03c94cd

20 июня 2017 г., 20:36:44 Temon137: Update classes.md f1816ca

20 июня 2017 г., 19:46:46 Temon137: Update basic-types.md 5a5aef6

20 июня 2017 г., 19:44:34 Temon137: Update visibility-modifiers.md 5bf2a7a

20 июня 2017 г., 19:44:19 Temon137: Update typecasts.md ff187ca

20 июня 2017 г., 19:44:05 Temon137: Update this-expressions.md 5d37357

20 июня 2017 г., 19:43:45 Temon137: Update returns.md 2c153cf

20 июня 2017 г., 19:43:27 Temon137: Update object-declarations.md dd12db4

20 июня 2017 г., 19:43:11 Temon137: Update nested-classes.md d5d76b6

20 июня 2017 г., 19:42:55 Temon137: Update inline-functions.md 0cec6a6

20 июня 2017 г., 19:42:32 Temon137: Update idioms.md 6748699

20 июня 2017 г., 19:42:22 Temon137: Update generics.md dfd31c1

20 июня 2017 г., 19:42:10 Temon137: Update functions.md bc3e2aa

20 июня 2017 г., 19:42:00 Temon137: Update control-flow.md e097b45

20 июня 2017 г., 19:41:49 Temon137: Update collections.md 473092a

20 июня 2017 г., 19:41:28 Temon137: Update classes.md 1ced9ea

20 июня 2017 г., 19:41:20 Temon137: Update basic-types.md 48eff4f

20 июня 2017 г., 19:07:17 Temon137: Update INDEX.md Убран пункт "Статьи, требующие проверки". 1a8e21d

20 июня 2017 г., 19:02:01 Temon137: Update operator-overloading.md 7a609df

20 июня 2017 г., 19:01:43 Temon137: Update functions.md d2e3a41

20 июня 2017 г., 14:48:30 Temon137: Update coroutines.md Исправления текста с целью улучшения читаемости. Некоторые исправления неточного перевода. 6ec6968

20 июня 2017 г., 11:30:06 Temon137: Update operator-overloading.md Добавлен отсутствующий пример. b998622

20 июня 2017 г., 11:03:31 Temon137: Update how-to-edit.md c853735

20 июня 2017 г., 10:59:56 Temon137: Update comparison-to-java.md 1002928

20 июня 2017 г., 10:56:52 Temon137: Update type-safe Строки 246-254: переведены непереведённые строки. 5bbfb94

20 июня 2017 г., 10:50:36 Temon137: Update reflection.md 0b2d4ce

20 июня 2017 г., 10:50:09 Temon137: Update annotations.md 534fb45

20 июня 2017 г., 10:05:03 Temon137: Исправление примечания 03a9015

20 июня 2017 г., 10:02:18 Temon137: Update typecasts.md 02594ed

20 июня 2017 г., 10:02:02 Temon137: Update this-expressions.md e523e01

20 июня 2017 г., 10:01:46 Temon137: Update operator-overloading.md 1e14ef2

20 июня 2017 г., 10:01:26 Temon137: Update null-safety.md aee33af

20 июня 2017 г., 10:01:07 Temon137: Update exceptions.md 924e2a3

20 июня 2017 г., 9:13:32 Temon137: Вторая попытка исправления комментариев 59f2f98

20 июня 2017 г., 9:11:50 Temon137: Update ranges.md 554f1b2

20 июня 2017 г., 9:11:36 Temon137: Update multi-declarations.md 88f44af

20 июня 2017 г., 9:11:11 Temon137: Update collections.md bf66985

20 июня 2017 г., 8:43:23 Temon137: Попытка исправления комментария Почему-то не скрывается закомментированный английский текст. 22a9574

20 июня 2017 г., 8:39:37 Temon137: Update coroutines.md 26b6368

20 июня 2017 г., 8:39:10 Temon137: Update inline-functions.md 13dd219

20 июня 2017 г., 8:38:08 Temon137: Update properties.md 0ca9bf5

19 июня 2017 г., 20:42:22 Temon137: Update lambdas.md 9d81895

19 июня 2017 г., 20:15:40 Temon137: Update visibility-modifiers.md 1f5acb8

19 июня 2017 г., 20:15:25 Temon137: Update sealed-classes.md 8ac43bd

19 июня 2017 г., 20:15:13 Temon137: Update object-declarations.md ac20b5b

19 июня 2017 г., 20:14:58 Temon137: Update generics.md b7a0e9a

19 июня 2017 г., 20:14:46 Temon137: Update functions.md 4eed3bf

19 июня 2017 г., 20:14:32 Temon137: Update extensions.md 32db813

19 июня 2017 г., 20:14:21 Temon137: Update enum-classes.md e4157fd

19 июня 2017 г., 20:14:00 Temon137: Update delegated-properties.md Исправления ошибок и удаление дублированного предлоэения (строка 140). a159972

19 июня 2017 г., 20:12:57 Temon137: Update data-classes.md 1ccf25c

19 июня 2017 г., 15:15:32 phplego: Merge pull request #36 from Temon137/patch-13 Update properties.md 7811f27

19 июня 2017 г., 15:14:13 phplego: Merge pull request #37 from Temon137/patch-12 Update classes.md a7a64ad

19 июня 2017 г., 15:13:01 phplego: Merge pull request #38 from Temon137/patch-11 Update returns.md 384ab30

19 июня 2017 г., 15:12:29 phplego: Merge pull request #39 from Temon137/patch-10 Update control-flow.md f23b741

19 июня 2017 г., 15:11:20 phplego: Merge pull request #40 from Temon137/patch-9 Update packages.md af5d945

19 июня 2017 г., 15:10:15 phplego: Merge pull request #41 from Temon137/patch-8 Update basic-types.md bb5a657

19 июня 2017 г., 15:06:57 phplego: Merge pull request #45 from Temon137/patch-14 Update interfaces.md 90dd7c6

19 июня 2017 г., 15:06:42 phplego: Merge pull request #43 from Temon137/patch-6 Update idioms.md fd94e0c

19 июня 2017 г., 15:05:14 phplego: Merge pull request #42 from Temon137/patch-7 Update coding-conventions.md 1c03e40

19 июня 2017 г., 15:04:25 phplego: Merge pull request #44 from Temon137/patch-5 Update basic-syntax.md db53111

19 июня 2017 г., 15:00:06 Temon137: Update interfaces.md c815ad5

19 июня 2017 г., 14:52:53 Temon137: Update basic-syntax.md 1779c95

19 июня 2017 г., 14:42:17 Temon137: Update properties.md 592d6a5

19 июня 2017 г., 14:33:01 Temon137: Update classes.md 7f830f6

19 июня 2017 г., 14:17:27 Temon137: Update returns.md dc58216

19 июня 2017 г., 14:11:11 Temon137: Update control-flow.md da1baef

19 июня 2017 г., 14:03:59 Temon137: Update packages.md 8fd980a

19 июня 2017 г., 14:01:34 Temon137: Update basic-types.md b4c2a75

19 июня 2017 г., 13:50:39 Temon137: Update coding-conventions.md 4c51ef5

19 июня 2017 г., 13:46:03 Temon137: Update idioms.md 8e01061

16 июня 2017 г., 20:20:23 phplego: убран пункт "прочая работа" 00c981e

13 июня 2017 г., 12:40:11 phplego: Create INDEX.md b06ba8e

13 июня 2017 г., 12:37:57 phplego: Merge pull request #35 from y2k/master Добавить ссылку на @kotlin_lang телеграм чат4bf55df

13 июня 2017 г., 10:46:19 y2k: add @kotlin_lang telegram chat 21e9fa9

12 июня 2017 г., 11:59:55 phplego: Merge pull request #33 from Temon137/patch-3 Create multi-declarations.md 2109a11

12 июня 2017 г., 11:51:55 phplego: Create functions.md 046be8d

12 июня 2017 г., 11:34:13 phplego: Create null-safety.md 982559a

12 июня 2017 г., 11:31:40 phplego: Merge pull request #34 from Temon137/patch-4 Исправление орфографических ошибок.1a065f9

12 июня 2017 г., 11:28:39 phplego: Merge pull request #32 from Temon137/patch-2 Исправления орфографических ошибок.7efcf80