



# **Язык программирования Kotlin (обзор)**

**Рассказчик:  
Михаил В. Шохирев**

**Клуб программистов  
2021-2025**



# О чём пойдёт речь?

- **Язык:**
  - **История** создания языка Kotlin. *Когда?*
  - Его **преимущества** и причины популярности. *Зачем?*
  - **Сравнение** языка Kotlin с *Java* и другими языками. *Почему?*
  - Установка. REPL. Скрипты. IDEA IDE. *Как?*
- **Синтаксис:**
  - Данные. Операции. Конструкции. Похожести и особенности. Идиомы.
- **Парадигмы:**
  - Функциональное программирование (**FP**).
  - Реализация **ООП** в Kotlin.
  - **Параллелизм**: сопрограммы, каналы, последовательности.
- **Применение:**
  - Kotlin **Multiplatform**. Compose Multiplatform.
    - Kotlin / JVM @ server & desktop.
    - Kotlin / JS @ server & browser, Kotlin / Wasm @ browser.
    - Kotlin / **Native** @ mobile devices & IoT (Raspberry Pi).
  - Kotlin для **Android**. KMM.



# Зачем учить **Kotlin**?

- **Java** широко применяется, а **Kotlin** – лучшая и полная замена **Java**.
- Программы на **Kotlin** можно совмещать с **Java** в одном проекте.
- Программировать на **Kotlin** проще, быстрее и эффективнее, чем на **Java**.
- Можно постепенно переводить кодовую базу с **Java** на **Kotlin**.
- Программы на **Kotlin** компилируются в исполняемый код (Kotlin/Native).
- **Kotlin** – многоплатформенный: Windows, MacOS, Linux, Raspberry Pi OS, ...
- На **Kotlin** разрабатываются мобильные приложения для Android и iOS.
- **Kotlin** применяется для "full-stack" разработки web-приложений.
- Для **Kotlin** есть первоклассная бесплатная IDE – IntelliJ IDEA.
- Программировать на **Kotlin** удобно, приятно и эффективно.
- Знаете другой популярный язык, разработанный в России, кроме **Kotlin**?

**Kotlin** – современное мощное средство разработки.



# Язык программирования



**Kotlin** — современный перспективный статически типизированный, объектно-ориентированный язык программирования. Исходники на **Kotlin** компилируются в JVM byte code, а также в JavaScript (Kotlin/JS) и в исполняемый код (Kotlin/Native) разных платформ и ОС через инфраструктуру LLVM.

*Язык разрабатывается компанией **JetBrains** и назван в честь острова Котлин в Финском заливе, на котором расположен город Кронштадт.*

У разработчиков **Kotlin**, которые создавали язык для себя, было 2 главных цели: (1) сделать язык более лаконичный и безопасный, чем Java, с более выразительным и удобным современным синтаксисом; и (2) обеспечить полную совместимость с огромной кодовой базой на Java.

**Kotlin** испытал влияние языков: Groovy, C#, Gosu, Java, Ruby, JavaScript, Scala, Python, ML, Pascal, G, C++, Rust, D.



# Преимущества

**Kotlin** разрабатывался как прагматичный язык для эффективного и удобного решения реальных задач с учётом современных требований.

**Kotlin** поддерживает разные *парадигмы* программирования: императивное, объектно-ориентированное, обобщённое, функциональное и другие.

**Kotlin** полностью совместим с **Java**: разработчики могут постепенно и безболезненно переходить к его использованию.

**Kotlin** адаптирован для разработки под **Android**: можно дорабатывать существующие приложения уже на Kotlin (без их переписывания) и создавать новые приложения.



# Сравнение

**Kotlin** лучше языка **Java**: у него компактный современный синтаксис, он удобнее и надёжнее, на нём быстрее и приятнее писать программы.

**Kotlin** — чисто объектно-ориентированный язык: в нём всё является объектами. В этом он логичнее гибридных языков типа **C++** или **Free Pascal**.

В **Kotlin** реализованы мощные средства функционального программирования: в этом он похож на языки **Scala**, **Scheme**, **Clojure**, **Lisp**.

**Kotlin** — строго типизированный язык, поэтому он надёжнее динамических языков наподобие **Perl**, **Ruby** или **Python**.

Программы на **Kotlin** могут компилироваться в «родной» двоичный исполняемый код для Intel / AMD или ARM-архитектур, поэтому он быстрее языков, основанных на байт-коде, вроде **C#**, **Erlang**, **Lua**.

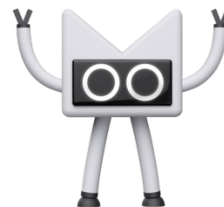
В **Kotlin** эффективно реализованы сопрограммы, в **Java** они только разрабатываются, но поддерживаются в **Ruby**, **Go**, **Rust** (1.39+), **Haskell**, **C#** (2.0+), **Python** (3.5+), **PHP** (5.5+) и других.



# История



**2010** – начало разработки языка под руководством *Андрея Бреслава*.  
2011.07 – язык Kotlin представлен общественности.  
**2012.02** – открыт исходный код реализации языка (Apache 2.0 Licence).  
2012.02 – выпущен milestone 1 с плагином для IDEA.  
2012.06 – milestone 2 с поддержкой Android.  
2012.12 – milestone 4 с поддержкой Java 7.  
**2016.02.15** – вышел официальный релиз **1.0**.  
2017.03.01 – вышел релиз 1.1.  
**2017.05** – Google включила поддержку Kotlin в Android Studio 3.0.  
2017.11.28 – Kotlin 1.2 с поддержкой Kotlin/JS.  
2018.10.29 – Kotlin 1.3 с поддержкой coroutines, asynchronous programming.  
**2019** – на Google I/O Kotlin объявлен приоритетным в разработке под Android.  
2020.08 – Kotlin 1.4 с поддержкой платформ Apple (Objective-C / Swift interop).  
**2020.11** – А. Бреслав уходит из JetBrains, разработку возглавил *Роман Елизаров*.  
2021.05 – Kotlin 1.5: JVM records support, stable JVM IR backend, Kotlin/JS IR backend.  
2021.11 – Kotlin 1.6: Kotlin/JS: IR compiler & Kotlin/Native performance improvements.  
2022.06 – Kotlin 1.7 с поддержкой alpha-версии нового компилятора K2.  
**2022.12.28** – в Kotlin **1.8** улучшена совместимость с Objective-C / Swift.  
**2023.07.06** Kotlin 1.9: улучшения для Kotlin Multiplatform и Kotlin/Native; K2 в beta.  
**2024.05.21** – Kotlin **2.0.0**.  
2024.11.27 – Kotlin 2.1.0.





# Почему популярен **Kotlin**?



- Создатель **Kotlin**, компания **JetBrains**, имеет международную репутацию разработчика первоклассных программных продуктов.
- **Kotlin** – это современный, более удобный, эффективный и надёжный язык на замену языка **Java**, на 100% совместимый с ним по байт-коду.
- **Google** сделала **Kotlin** приоритетным средством разработки для **Android**, интегрировала его в свои продукты (Android Studio, SDK, KTX).
- **Kotlin** предоставляет средства кроссплатформенной разработки Kotlin Multiplatform [Mobile] и Kotlin/Native.
- **Kotlin** — современный язык, в котором применены достижения из ранее разработанных языков и сегодняшние требования к разработке.





# Kotlin: установка, компиляция

0. Установить JDK, если не ставить IDE.

1. Установить **Kotlin** одним из способов:

- Установить IDE **IntelliJ IDEA CE**.
- Установить IDE **Android Studio**.
- Установить **SDKman** (можно под WSL или CygWin):

```
curl -s "https://get.sdkman.io" | bash
```

и выполнить:

```
sdk install kotlin
```

- Под GNU / Linux выполнить:

```
sudo snap install --classic kotlin
```

- На macOS выполнить:

```
brew install kotlin
```

2. Сохранить исходную программу в файле, например: *hello.kt*.

3. Скомпилировать в байт-код: **kotlinc** *hello.kt* -include-runtime -d *hello.jar*

4. Выполнить:

```
kotlin hello.jar Миру мир
```

или:

```
java -jar hello.jar Миру мир
```



# Kotlin: REPL

```
t4503@PC973:~/D/_Learn/languages $ kotlinc
Welcome to Kotlin version 1.8.0 (JRE 11.0.16+8-post-Debian-1deb10u1)
Type :help for help, :quit for quit
>>> val s = "Строка"
>>> s.toUpperCase()
res1: kotlin.String = СТРОКА
>>> s.length
res2: kotlin.Int = 6
>>> :quit
t4503@PC973:~/D/_Learn/languages $
```

Можно изучать **Kotlin** с помощью интерактивной среды (по типу REPL = Read-Eval-Print Loop), которая построчно исполняет команды, вводимые с консоли (в терминале / командном окне). Для этого запустить компилятор без параметров:  
**kotlinc**

*(Совсем как у таких скриптовых языков, как Ruby или Python.)*

**<https://play.kotlinlang.org/>** — online interactive Kotlin playground!



# Kotlin: скрипты

**Kotlin** можно использовать как скриптовый язык.

Для этого в файл с суффиксом **.kts** записывается упрощённая исходная программа на языке Kotlin, в которой даже нет функции `main()`, например, 1 строка:

```
println("Мир — мир!")
```

Имя файла с исходником скрипта передаётся компилятору после параметра `-script` на компиляцию и немедленное выполнение:

```
kotlinc -script script_name.kts
```



# hello.kt

```
const val LANGUAGE = "Kotlin" // константа глобальная
/* подпрограмма */
fun prepare(head: String,
            tail: String = LANGUAGE): String {
    return "$head, $tail!"
}
/** Doc comment:
 * главная программа
 * @param args – аргументы из командной строки
 */
fun main(args: Array<String>) {
    val greeting = "Привет" // неизменяемая переменная
    var message: String = // изменяемая переменная
        if (args.size == 2) { // условное выражение
            prepare(args[0], args[1])
        } else {
            prepare(greeting, LANGUAGE)
        }
    println(message)
}
```

Синтаксис **Kotlin** –  
выразительный,  
ясный и понятный,  
легко осваивается  
любым, кто знает  
такие языки, как  
C, C#, Java,  
Objective-C, Swift,  
Ruby, Python  
или другой  
ОО-язык.



# Kotlin vs Java

```
const val LANGUAGE = "Kotlin"
```

```
fun prepare(head: String = "Привет",  
            tail: String = LANGUAGE): String {  
    return "$head, $tail!"  
}
```

```
fun main(args: Array<String>) {  
    val message: String =  
        if (args.size == 2) {  
            prepare(args[0], args[1])  
        } else {  
            prepare()  
        }  
    println(message)  
}
```

```
public final class HelloKt {
```

```
    public static final String LANGUAGE = "Kotlin";
```

```
    public static final String prepare(String head, String tail) {  
        return head + ", " + tail + '!';  
    }
```

```
    public static final void main(String[] args) {  
        String greeting = "Привет";  
        String message = null;  
        if (args.length == 2) {  
            message = prepare(args[0], args[1]);  
        } else {  
            message = prepare(greeting, LANGUAGE);  
        }  
        System.out.println(message);  
    }  
}
```

Короче (на ~40%), проще, нагляднее:  
легче писать, только необходимое.  
*Встроенные проверки на null.*

Многословнее, тяжелее, хуже читается:  
надо больше писать, чаще можно ошибиться.  
Потребуются дополнительные проверки на null.



# Синтаксис: именование

// принятые в Kotlin соглашения об именах:

// классы

val truth: **Boolean** = true

// с заглавной буквы

// литералы числовые

+1.23E-4, 42UL

// + - 0-9 . \_ b B x X f F u U UL e E

// константы

const val **LANGUAGE** = "Kotlin"

// заглавными буквами и "\_"

// переменные

var **languageAuthor** = "Андрей Бреслав" // CamelCase

// функции

fun **yearOfCreation**(): Int { return 2010 } // CamelCase



# Данные: литералы

**Boolean:** true, false

**Byte:** -1, 0b0101, 0x0A1.toByte() // целые со знаком

**Int:** 0, -25, 42, 12345, 0b1111\_1111\_1111\_1111\_1111\_1111

**Short:** 0, -32768, 4\_2, 1\_000\_000

**Long:** 1\_000\_000L, 1\_000\_000\_000\_000\_000\_000

// беззнаковые целые

**UByte:** 0b0, 0xF1, 5u

**UShort:** 0b0, 65535, 0xFFFF, 25U

**UInt:** 0, 4\_294\_967\_295, 1000000u

**ULong:** 18\_446\_744\_073\_709\_551\_615, 42UL

// восьмеричных литералов нет

**Float:** 0.0001F, -0.25F, 4.2f

**Double:** 1000.0123456789, 42.0, 23e-3

**Char:** 'R', '#', 'Ш', '0', '\t', '\n', '\r', '\'', '\"', '\\'

**String:** "Хочу выучить Kotlin", "42", "R", "今日は!", ""Много "\n" строк""

Литералы  
объявляются  
практически  
так же, как  
во многих  
других языках,  
начиная с  
Perl  
или даже  
C.



# Данные: ТИПЫ ДАННЫХ

```
val identifier [: Type] = initialization // ссылка на объект неизменна  
var identifier: Type [= initialization] // ссылка на объект изменяема
```

```
const val LANGUAGE = "Kotlin" // константа с «авто-типом»  
val truth: Boolean = true // «неизменяемая» переменная с типом  
var lie = false // «изменяемая» переменная  
var statement: Boolean? // Nullable Boolean: true|false|null  
  
val character: Char = 'R' // 2 bytes = 16 bits, UTF-16 как в Java  
  
val string = "JetBrains" // вывод типа = type inference  
var text: String // объявление типа без значения  
  
var variable: Any = "Kotlin" // ≈ Object @ Java  
variable = 2010  
  
if (obj is ClassName) { /* проверка типа = принадлежность к классу */ }  
if (obj !is ClassName) { /* не принадлежит к классу? */ }
```





# Данные: числа

```
// Signed integer numbers
val byte: Byte = -5 // 8 bits = 1 byte: -128..127
val short: Short = 25 // 16 bits = 2 bytes: -32768..32767
val integer: Int = 42 // 32 bits = 4 bytes: -231..231-1
val long: Long = 32_768L // 64 bits = 8 bytes: -263..263-1

// Unsigned integer numbers
val unsigned08: UByte = 32U // 8 bits = 1 byte: 0..255
val unsigned16: UShort = 128U // 16 bits = 2 bytes: 0..65535
val unsigned32: UInt = 256U // 32 bits = 4 bytes: 0..232-1
val unsigned64: ULong = 32_768U // 64 bits = 8 bytes: 0..264-1 байт

// Floating point fractional numbers
val pi: Float = 3.1415926F // 4 байта: -3.4*1038..3.4*1038
val height: Double = -23e3 // 8 байт: ±5.0*10-324..±1.7*10308
var one = 0.0f; repeat(10) { one += 0.1f } // 1.0000001
```



# Данные: СИМВОЛЫ

```
val character: Char = 'R'           // 2 bytes = 16 bits (UTF-16)
val at = '\u0040'                   // '@'
val backslash = '\\'                 // '\'
val a = 97.toChar()                  // 'a'
var ch = a + 2                       // 'c'
val b = ch - 1                       // 'a'
    ch -= 24                         // 'K'
    println(++ch)                    // 'L'
    println('a' < 'c')               // true
'\n'.isWhitespace()                 // true
Character.isWhitespace(ch)           // false
'k'.uppercaseChar(); 'k'.uppercase() // 'K', "K"

val ch: Char = "Kotlin".first()      // 'K'
println("Kotlin"[2])                 // 't'
println(""+'a'+ 'b'+ 'c')            // "abc"
```



# Данные: строки

```
var string = "Java" // strings are immutable
string = "Kotlin"   // ссылка на новую строку

val character = string[0] // 'K' – 0-я буква в строке
string.forEach { char -> print(char) } // итератор по символам
for (char in string) { print(char) }   // цикл по символам

string + " – язык программирования № "+ 1 + "!" // операция '+' (сцепление)
"$string разработан в ${2*1000+10} году"         // подстановка выражения

string.length; string.lastIndex; // свойства строк
"Kotlin".substring(0, 3)           // "Kot"
"true".toBoolean()                // методы строк
"3.14".toDouble(); "42".toInt(); "25000".toLong();
string.uppercase()
string.substring(3)                // "lin"
string.replace("Kotlin", "Java", true) // true = ignore case
string.isEmpty()                   // false
"*".repeat(6)                     // "*****"
```



# Данные: <обобщения>

**Обобщения** (Generics) — вариантность на уровне объявления (модификаторы):

- **in** — параметризованный тип *контравариантен* (contravariant): он может только потребляться, но не может производиться (Consumer in);
- **out** — параметризованный тип *ковариантен* (covariant): он только возвращается (производится) и никогда не потребляется (Producer out);
- **where** — в <> может быть указана только одна верхняя граница, для указания нескольких верхних границ нужно использовать отдельное условие where. Переданный тип должен одновременно удовлетворять всем условиям where. В приведённом примере тип T должен реализовывать и CharSequence, и Comparable.

```
fun <T> copyWhenGreater(list: List<T>, threshold: T): List<String>
    where T : CharSequence, T : Comparable<T> {
    return list.filter { it > threshold }.map { it.toString() }
}
```



# Синтаксис: особенности

```
var `Это такое длинное имя переменной с пробелами на русском языке`: String

fun sideEffect(): Unit {} // ≈ void, точнее: некий

fun doNothingForever(): Nothing { // ничего не возвращает и
    while(true) { } // никогда не завершается
} // void @ Java

fun maxOf(a: Int, b: Int) = if (a > b) a else b // тело функции - выражение

var value: Int? = null // Nullable-тип

val string: String? = "Не пусто" as? String // "Не пусто" || null

var (a, b) = listOf(1, 2) // параллельное присваивание
a = b.also { b = a } // обмен значений переменных

fun String.spaceToCamelCase() { /* ... */ } // функции-расширения для классов:
"Convert this to camelcase".spaceToCamelCase()
```



# Синтаксис: операции

Оператор присваивания '=' в Kotlin – *инструкция*, а в Java – *выражение*.

Арифметические:

`+` `-` `*` `/` `%` `++` `--` `+=` `-=` `*=` `/=` `%=`

Поразрядные:

`and` `or` `xor` `inv` `shl` `shr` `ushr`

Отношения:

`==` `!=` `<` `>` `>=` `<=` `===` `!==`

Логические:

`&&` `||` `xor` `!` `not()` `and` `or` `in` `!in`

Строковые:

```
var message = "Привет, " + "Kotlin"
```

// конкатенация

```
message += '!'
```

```
"1/3 равно ${1.0 / 3.0} в виде десятичной дроби."
```

// подстановка

Операции  
практически  
такие же,  
как в других  
языках  
программирования.



# Синтаксис: operator overloading

## Оператор

+a  
-a  
!a  
a + b  
a - b  
a \* b  
a / b  
a % b

## Метод

a.unaryPlus()  
a.unaryMinus()  
a.not()  
a.plus(b)  
a.minus(b)  
a.times(b)  
a.div(b)  
a.rem(b)

a += b  
a -= b  
a \*= b  
a /= b  
a %= b

a.plusAssign(b)  
a.minusAssign(b)  
a.timesAssign(b)  
a.divAssign(b)  
a.remAssign(b)

a..b  
a in b  
a !in b

a.rangeTo(b)  
b.contains(a)  
!b.contains(a)

## Оператор

a == b  
a != b  
a > b  
a < b  
a >= b  
a <= b

++a  
--a

a[i]  
a[i, j]  
a[i\_1, ..., i\_n]  
a[i] = b  
a[i, j]=b  
a[i\_1, ..., i\_n]=b

a()  
a(i\_1, ..., i\_n)

## Метод

a?.equals(b) ?: (b === null)  
!(a?.equals(b) ?: (b===null))  
a.compareTo(b) > 0  
a.compareTo(b) < 0  
a.compareTo(b) >= 0  
a.compareTo(b) <= 0

a.inc()  
a.dec()

a.get(i)  
a.get(i, j)  
a.get(i\_1, ..., i\_n)  
a.set(i, b)  
a.set(i, j, b)  
a.set(i\_1, ..., i\_n, b)

a.invoke()  
a.invoke(i\_1, ..., i\_n)



# Синтаксис: ветвление

```
val y = if (x == 1) {  
    "one"  
} else if (x == 2) {  
    "two"  
} else {  
    "other"  
}
```

// if-else возвращает результат

```
fun describe(obj: Any): String =  
    when (obj) {  
        1 -> "Один"  
        "Hello" -> "Приветствие"  
        is Long -> "тип Long"  
        !is String -> "Не строка"  
        else -> "UNKNOWN"  
    }
```

// when возвращает результат





# Синтаксис: ЦИКЛЫ

```
val list = listOf("яблоко", "банан", "киви", "фига", "авокадо")
```

```
for (item in list) { print(item) }  
for (i in list.indices) { print(list[i]) }
```

```
var i = 0  
while (i < list.size) {  
    print(list[i]); i++  
    if (i % 2 == 0) { continue }  
    if (i == 2) { break }  
}
```

```
for (i in 1..n step 2) { print(i) }  
for (i in n downTo 0 step 3) { print(i) }
```

```
repeat(n) { print(message) }
```

```
var i: Int = 0  
do {  
    print(list[i])  
    i++  
} while (i < list.size)  
// post-test loop
```

```
print(message.repeat(n))
```



# Синтаксис: диапазоны

```
val range = m..n // >=m, <=n
if (x in range) { print("$x – в диапазоне от $m до $n") }

println("hello" in "he".. "hi") // true
if (character in 'a'..'z') { print("Маленькая латинская буква") }

for (i in 1..n+1) { print(i) } // i >= 1 && i <= n+1

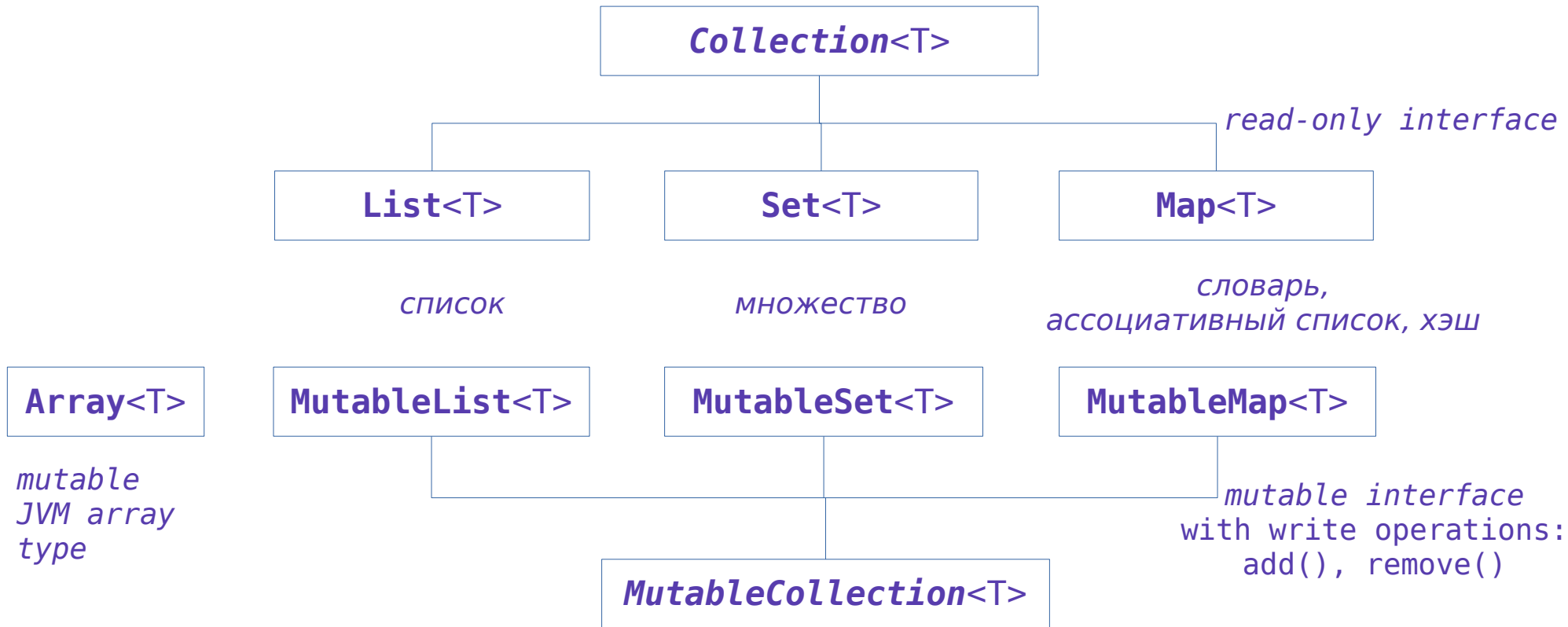
for (i in list.indices) { print(list[i]) }

for (i in 1..100) { print(i) } // закрытый диапазон: <= 100
for (i in 1 until 100) { print(i) } // полуоткрытый диапазон < 100
for (i in 2..10 step 2) { print(i) }
for (i in 10 downTo 1) { print(i) }

(1..10).forEach { print(message) } // итератор по диапазону
```



# Синтаксис: коллекции







# Синтаксис: изменяемые списки

```
var mutableList = mutableListOf("日", "月")
mutableList += "星" // mutableList.add("星")
mutableList -= "月" //
mutableList.removeAt(1)
print(mutableList[1])
mutableList.addAll(anotherMutableList) // join
val newMutableList = mutableList + anotherMutableList // concatenate
val mutableListCopy = mutableList.toMutableList() // copy

val numbers = mutableListOf<Int>(1, 2, 3, 4, 5)
numbers.removeAt(0) // == numbers.remove(1)
numbers.add(25) // append
numbers.add(0, 11) // prepend
println(numbers.shuffle().joinToString(";")) // 2; 25; 3; 4; 5; 11
numbers.last() == numbers[numbers.size-1] == numbers[numbers.lastIndex]
numbers.clear()
```



# Синтаксис: многомерные списки

```
// multi-dimensional list == list of lists

val mutableList2D = mutableListOf(
    mutableListOf<Int>(2),           // 2D list == matrix columns ==
    mutableListOf<Char>('B', 'O'),  // таблица из 3 строк по N колонок
    mutableListOf<String>("not", "to", "be") // [0][0]
)                                   // [1][0],[1][1]
print(mutableList2D[2].joinToString()) // [2][0],[2][1],[2][2]

manyDimensionalList[latitude][longitude][height][temperature][humidity][pressure]
```



# Синтаксис: словари

```
// Map<K, V> // ассоциативный список, словарь, хэш
var map = mapOf("Kotlin" to 2016, "Java" to 1995, "Ruby" to 1995)
    map = mapOf(Pair("Kotlin", 2016), Pair("Java", 1995), Pair("Ruby", 1995))
map["Kotlin"] // == map.get("Kotlin")
map -= "Java"

map.entries // свойства: список пар
map.keys // список ключей
map.values // список значений
map.size // размер
map.count { it.value == 1995 } // методы: подсчёт
map.containsKey("Ruby") // поиск ключа
map.containsValue(1995) // поиск значения
map.toSortedMap() // сортировка по ключам
map.filter{ it.key == "Kotlin" || it.value == 1995 } // фильтр

for ((k, v) in map) { print("$k => $v") } // цикл по Map
map.forEach { k, v -> print("$k => $v") } // итератор по Map
```



# Синтаксис: изменяемые словари

```
// MutableMap<K, V>
// var map: MutableMap<Any, Any> = mutableMapOf()
var mutableMap = mutableMapOf("Kotlin" to 2016, "Java" to 1995, "Ruby" to 1995)

mutableMap.put("Python", 1991)
mutableMap["Perl"] = 1987
mutableMap += Pair("JavaScript", 1995)
mutableMap += "Swift" to 2014

mutableMap.remove("Java")
mutableMap -= "Python"
```

По умолчанию `MutableMap` реализован как **LinkedHashMap**, который сохраняет порядок добавления элементов.

Другая реализация, **HashMap**, не сохраняет порядок добавления элементов.





# Синтаксис: множества

```
// Set<T>
```

```
val set = setOf("一", "二", "三", "一")
set.first()
set.last()
set.elementAt(1)
setOf(1,2,3) union setOf(4,5)
setOf(1,2,3) intersect setOf(2,1)
setOf(1,2,3) subtract setOf(4,3)
```

```
// дубликат не попадёт в набор
// "一", "二", "三"
// "一"
// "三"
// "二"
// слияние: [1,2,3,4,5]
// пересечение: [1,2]
// вычитание: [1,2]
```

```
val listWithDuplicates = listOf("四", "五", "六", "七", "五")
val setWithoutDuplicates = list.toSet()
```

```
// "四", "五", "六", "七"
```

```
// MutableSet<T>
```

```
var mutableSet = mutableSetOf("Red")
mutableSet += "Yellow"
mutableSet.add("Blue")
mutableSet -= "Red"
```



# Kotlin: functional programming

- Глобальные функции: вне классов.
- Вложенные функции (*local functions*).
- Функции-расширения.
- Тип данных: функция.
- Лямбды (*Lambda*).
- Функции высшего порядка (*higher order function*).
- Функции среды (*scope functions*).
- Удобный синтаксис:
  - функции-выражения (*expression body*);
  - аргументы по умолчанию (*default arguments*);
  - именованные аргументы (*named arguments*);
  - инфиксная нотация функций (*infix functions*);
  - приём произвольного количества аргументов (*vararg*);
  - передача одного аргумента lambda без ()



# Синтаксис: функции

```
fun functionName(p1: Type1, p2: Type2, ...): ReturnType {  
    // body  
    return result  
}
```

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

```
fun difference(a: Int, b: Int) = a - b                       // expression body
```

```
fun printSum(a: Int, b: Int): Unit {                         // не возвращает значения ≈ void  
    print("$a + $b == ${sum(a, b)}")  
}
```

```
fun printDifference(a: Int = 0, b: Int = 0) {                 // значения аргументов по умолчанию  
    print("$a - $b == ${difference(a, b)}")  
}
```

```
fun `login with correct login and password`() {              // имена функций для тестов  
    //given //when //then  
}
```



# Функции: аргументы

// аргументы по умолчанию (default arguments)

```
fun square(length: Int = 1, width: Int = length) = length*width
result = square(10, 20)           // 10*20 == 200
result = square(10)                // 10*10 == 100
result = square()                  // 1*1 == 1
```

// именованные аргументы (named arguments)

```
fun volume(length: Int, width: Int, height: Int=1) = length*width*height

result = volume(300, 20, 1)        // positional arguments
result = volume(length = 300, 20, 1) // Kotlin version >= 1.4
result = volume(width = 20, length = 300, 1)
result = volume(height = 1, length = 300, width = 20)

result = volume(300, 20)            // height=1 по умолчанию
result = volume(width = 20, length = 300)
```



# Функции: локальные

```
fun compareAge(age1: Int, age2: Int): Int? {  
    fun ageIsValid(age: Int) = age > 0 && age < 111           // вложенная функция  
  
    if( !ageIsValid(age1) || !ageIsValid(age2)) {  
        println("Invalid age")  
        return null  
    }  
  
    return when {  
        age1 < age2 -> -1  
        age1 == age2 -> 0  
        age1 > age2 -> 1  
    }  
}
```



# Функции-расширения

// Можно расширять существующие классы,  
// добавляя в них новые методы и свойства!

```
package strings  
fun String.lastCharter(): Char = this.get(this.length-1) // метод  
var StringBuilder.lastChar: Char                        // СВОЙСТВО  
    get() = get(length-1)  
    set(value: Char) { this.setCharAt(length-1, value) }
```

```
import strings.*  
print("Kotlin".lastCharacter())  
val string = StringBuilder("Kotlin?")  
string.lastChar = '!'  
print(string.lastChar)
```



# Данные: функции

```
// Тип данных – функция:  
    (arg1: Type1, arg2: Type2, ...) -> ReturnType  
  
val lambda = { print("Это лямбда ") }    // сохранить функцию в переменной  
lambda()                                // тип: () -> Unit  
  
val runAndLog: (() -> Unit) -> String = { // -> String = { func ->  
    it()  
    "... завершила работу"  
}  
  
println(runAndLog(lambda))              // "Это лямбда ... завершила работу"
```



# Функции: лямбды

// lambda-выражения для фильтрации и модификации коллекции в цепочке

```
val list = listOf("яблоко", "апельсин", "киви", "фига", "авокадо")
list.sortedBy { it }                // отсортировать
    .filter { it.startsWith("a") }  // выбрать
    .map { it.uppercase() }         // озаглавить
    .forEach { println(it) }        // => АВОКАДО АПЕЛЬСИН
```

// lambda – параметр функции

```
people.maxBy({ p: Person -> p.age }) // lambda как аргумент функции maxBy
people.maxBy() { p: Person -> p.age } // lambda можно вынести за скобки
people.maxBy { p: Person -> p.age }   // тип параметра указан явно
people.maxBy { p -> p.age }           // тип параметра выводится из контекста
people.maxBy { it.age }               // it – автоматически сгенерированное имя параметра
```





# Kotlin: object-oriented programming

Классы. Классы данных. Перечисления. Sealed class.

Class members:

- Конструкторы. Инициализирующие блоки.

- Функции (методы)

- Свойства. Accessors. Делегирование свойств.

- Вложенные классы.

- Вспомогательные объекты.

Наследование. Открытые классы. Абстрактные классы.

Интерфейсы. Функциональные интерфейсы (SAM interfaces).

Объекты. Анонимные объекты (object expressions).



# ООП: класс данных

```
// DTO = POJO = POCO
// структура для данных с описанием
data class Book(
    var title: String, // свойств
    var author: Author // (class fields)
)
```

Создаёт класс *Book* с такими возможностями:

- геттеры (и сеттеры в случае var) для всех СВОЙСТВ,
- метод equals(),
- метод hashCode(),
- метод toString(),
- метод copy(),
- методы component1(), ..., componentN() для всех СВОЙСТВ:

```
val (title, author) = book
```

```
/* Java*/
class Book
    private String title;
    private Author author;

    public String getTitle() {
        return title;
    }
    public void setTitle(String title){
        this.title = title;
    }

    public Author getAuthor() {
        return author;
    }
    public void setAuthor(Author author){
        this.author = author;
    }
}
```



# ООП: класс

```
class Person constructor (                                // параметры конструктора
    val givenName: String,                               // свойства (R)
    val familyName: String,                             //              (R)
    var birthday: LocalDate) {                          //              (RW)
    // блок инициализации #1

    init { println("Initializer block") }                // member function

    fun fullName(): String {
        return "$givenName $familyName"
    }                                                     // метод

    init { println(fullName()) }                         // блок инициализации #N
}

// вызов конструктора
val son = Person("Lev", "Shock", LocalDate.of(1988, 6, 24))
```



# ООП: конструкторы

*// constructor signature: количество, типы, порядок параметров*

```
class Size(                                     // primary constructor
    var width: Double = 1.0,
    var height: Double = 1.0) {
    var area: Double = width * height           // secondary constructors
    constructor(h: Int): this(height = h.toDouble()){ }
                                                // constructor delegation
    constructor(w: Int, h: Int): this(w.toDouble(), h.toDouble()) { }
    constructor(w: Int, h: Double): this(w.toDouble(), h) { }
    constructor(h: Double, w: Int): this(w.toDouble(), h)
    constructor(s: Size): this(s.width, s.height)
}
```

```
val size1 = Size(7)                           // 1st constructor
val size2 = Size(1, 7)                         // 2nd constructor
val size3 = Size(1, 7.0)                       // 3rd constructor
val size4 = Size(7.0, 1)                       // 4th constructor
val size5 = Size(size4)                       // 5th constructor
```



# Класс: accessors

```
class Someone(name: String) {           // не свойство, а параметр конструктора!
    var name: String = name              // явные:
    get() = field                        // getter
    set(value) {                          // setter
        previousName = field            // backing field of a property
        field = value
    }
    var previousName = "Unknown"

    var grammyAwards: Int = 0
    get() { field++; return field }      // свойство (вычисляемое)
}
```

```
val musician = Someone("David Robert Jones")
musician.name = "David Bowie"
println(musician.previousName)
println(musician.name)
repeat(5) { print("${musician.grammyAwards} ") }
```



# Класс: свойства

```
class Rectangle(  
    var height: Double = 1.0,           // свойство из конструктора  
    var width: Double = 1.0) {          // свойство из конструктора  
    val color: Int = 0xFFFFFFFF          // class property  
    var perimeter = (height + width) * 2 // свойство (вычисляемое)  
}  
  
// Инициализация свойств объекта с помощью функции apply  
val myRectangle = Rectangle().apply {  
    height = 2.0  
    width = 5.0  
    color = 0xFAFAFA  
} // Полезно для конфигурации свойств, которых нет в конструкторе объектов
```



# Класс: методы

```
class Turtle {  
    fun penDown()                // member functions  
    fun penUp()  
    fun turn(degrees: Double)  
    fun forward(pixels: Double)  
}  
  
val myTurtle = Turtle()          // default constructor  
with(myTurtle) {                 // вызов нескольких методов объекта (with)  
    penDown()  
    for (i in 1..4) {             // нарисует квадрат размером 100x100 px  
        forward(100.0)  
        turn(90.0)  
    }  
    penUp()  
}
```



# Класс: модификаторы доступа

4 модификатора доступа: `private`, `protected`, `internal` и `public` (по умолчанию):

- **public** — любой клиент, который видит объявленный класс, видит его `public`-члены.
- **private** означает видимость только внутри этого класса (включая его члены);
- **protected** — то же самое, что и `private` + видимость в подклассах;
- **internal** — любой клиент внутри модуля, который видит объявленный класс, видит и его `internal`-члены;

Если `protected`- или `internal`-член переопределен и явно не указана его видимость, то переопределенный элемент будет иметь тот же модификатор, что и исходный.





# ООП: наследование

```
open class Shape {  
    open fun draw() { /*...*/ }  
    fun fill() { /*...*/ }  
}  
                                     // пустой открытый класс  
  
class Rectangle(var height:Double=1.0, var width:Double=1.0): Shape() {  
    var perimeter = (height + width) * 2           // свойство (вычисляемое)  
    override fun draw() { /*...*/ }  
}  
  
val rectangle = Rectangle(5.0, 2.0)               // конструктор  
  
println("Периметр: ${rectangle.perimeter}")
```



# ООП: объекты

```
object Singleton { // named object is a singleton & cannot be local
    var name: String = "Это объект «одиночка»"
    fun printName() {
        println(name)
    }
}
print(Singleton.name)
Singleton.printName()

val helloWorld = object { // anonymous object
    val hello = "Hello"
    val world = "World"
    // тип анонимных объектов Any, поэтому необходим override в toString()
    override fun toString() = "$hello, $world!"
}
helloWorld.toString()
```



# ООП: перечисления

```
enum class Rainbow0 { // контейнер для набора констант
    RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
}

// инициализация элементов через передачу значений конструктору:
enum class Rainbow(val color: String, val rgb: String) {
    RED("Красный", "#FF0000"),
    ORANGE("Оранжевый", "#FF7F00"),
    YELLOW("Жёлтый", "#FFFF00"),
    GREEN("Зелёный", "#00FF00"),
    BLUE("Синий", "#0000FF"),
    INDIGO("Индиго", "#4B0082"),
    VIOLET("Фиолетовый", "#8B00FF"); // ; здесь обязательна!
    fun printFullInfo() { println("Color - $name, rgb - $rgb") }
}

val red: Rainbow = Rainbow.RED
println(red.color); red.printFullInfo()
print(Rainbow.RED.color); println(Rainbow.RED.rgb) // Красный#FF0000
println(Rainbow.BLUE.name); println(Rainbow.BLUE.ordinal) // BLUE4
for (enum in Rainbow.values()) { println(enum.name) }
println(Rainbow.valueOf("YELLOW")) // YELLOW
Rainbow.values().any { it.name == "GREEN" } // поиск в перечислении по имени
```



# Типы параллелизма

В различных языках реализованы разные подходы к многопоточному программированию:

- основанные на callback (JavaScript);
- основанные на future / promise (Java, JavaScript), deferred (Kotlin);
- основанные на async / await (C#, Kotlin);
- сопрограммы на state machines (Kotlin);
- сопрограммы на fibers (Ruby);
- goroutines на CSP (Go);
- actor model — форма cooperative multitasking;

Все они могут быть реализованы при помощи сопрограмм, так как не навязывают стиль программирования. С их помощью может быть реализован любой стиль многопоточного программирования, а некоторые уже реализованы в библиотеках Kotlin.



# Сопрограммы

**Сопрограммы** (coroutines) – это приостанавливаемые вычисления в виде легковесных потоков. Это реализация *кооперативной многозадачности* (cooperative multitasking), в которой мы сами управляем потоками (в отличие от вытесняющей многозадачности (preemptive multitasking) с потоками (Threads) ОС или JVM).

Сопрограммы в **Kotlin** оформляются как простой последовательный код, лёгкий для понимания, а вся сложность прячется внутри библиотек. Они предоставляют возможность запускать асинхронный код без блокировок. Вместо блокировки потоков вычисления становятся прерываемыми.

Сопрограммы работают как **конечные автоматы**, которые переходят из одного промежуточного состояния в другое, пока не достигнут конечного состояния.



# Сопрограммы: `suspend`

Сопрограммы оформляются в виде функций с модификатором **`suspend`** (т. н. «окраска функций»).

Компилятор добавляет в такие функции дополнительный параметр типа `kotlin.coroutines.Continuation`, а тип возвращаемого результата меняет на `kotlin.Any?`, т. к. `suspend`-функция может либо приостанавливаться (возвращать маркер состояния), либо возвращать результат.



# Сопрограммы: scope

Несколько сопрограмм из одного метода выполняются в «логической сфере» (scope), привязанной к жизненному циклу метода. При завершении метода все его сопрограммы отменяются, чтобы избежать утечки памяти. Нет возможности выйти из scope, пока все сопрограммы в нём не закончатся.

В **Kotlin** предусмотрен общий GlobalScope, а у каждой из suspend-функции — coroutineScope. If a coroutine is started on GlobalScope, the program will not wait for it. A parent provides a scope for its children,

Комбинируя «сферы» можно организовать дерево вызовов сопрограмм.

Этот подход называется «структурированный параллелизм» (structured concurrency).

- children inherit context from their parent (but they can also overwrite it, as will be explained in the Coroutine context chapter);
- a parent suspends until all the children are finished (this will be explained in the Job and children awaiting chapter);
- when the parent is cancelled, its child coroutines are cancelled too (this will be explained in the Cancellation chapter);

when a child raises an error, it destroys the parent as well (this will be explained in the Exception handling chapter).



# Сопрограммы: builder

Сопрограмма запускается с помощью одного «построителей» (coroutine builder):

- **launch** — запускает сопрограмму на независимое параллельное выполнение в виде легковесного процесса (через диспетчер потоков).
- **async** — похож на `launch`, но возвращает через `lambda`-выражение значение в виде объекта типа `Deferred<T>`, где `T` — это тип возвращаемого значения. У `Deferred`-объекта есть метод `await()`, который возвращает это значение, когда оно готово (при окончании работы сопрограммы).
- **runBlocking** — блокирует поток, из которого была запущена сопрограмма, когда сопрограмма приостанавливается: например, чтобы `main()` не закончилась раньше сопрограммы.





# Сопрограммы: async

```
// асинхронное (параллельное) выполнение
```

```
import kotlin.system.measureTimeMillis
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun doSomething(message: String): String {
    println("$message started")
    Thread.sleep(1000L) // задержка 1 сек.
    println("$message finished")
    return message
}

suspend fun main() {
    val time = measureTimeMillis {
        GlobalScope.async { doSomething("One") }
        GlobalScope.async { doSomething("Two") }
    }
    println("Elapsed: $time ms.")
}

// Elapsed: 36 ms.
// One started
// Two started
```

```
// async – это coroutine builder
// async() возвращает Deferred Job
// await() ожидает завершения сопрограммы
```

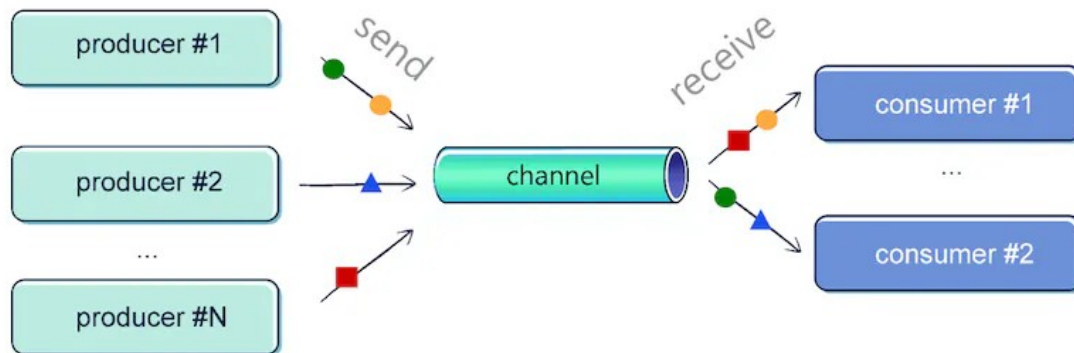
```
suspend fun main() {
    val time = measureTimeMillis {
        val j1 = GlobalScope.async { doSomething("One") }
        val j2 = GlobalScope.async { doSomething("Two") }
        j1.await() + j2.await()
    }
    println("Elapsed: $time ms.")
}

// One started
// Two started
// One finished
// Two finished
// Elapsed: 1035 ms.
```



# Сопрограммы: каналы

**Канал** (channel) подобен очереди (queue). У канала есть suspend-функции `send` и `receive`. Поэтому несколько сопрограмм-поставщиков (producer) могут записывать в канал, и несколько сопрограмм-потребителей (consumer) могут читать из этого канала, то есть использовать каналы, чтобы передавать данные друг другу без блокировки.



Виды каналов:

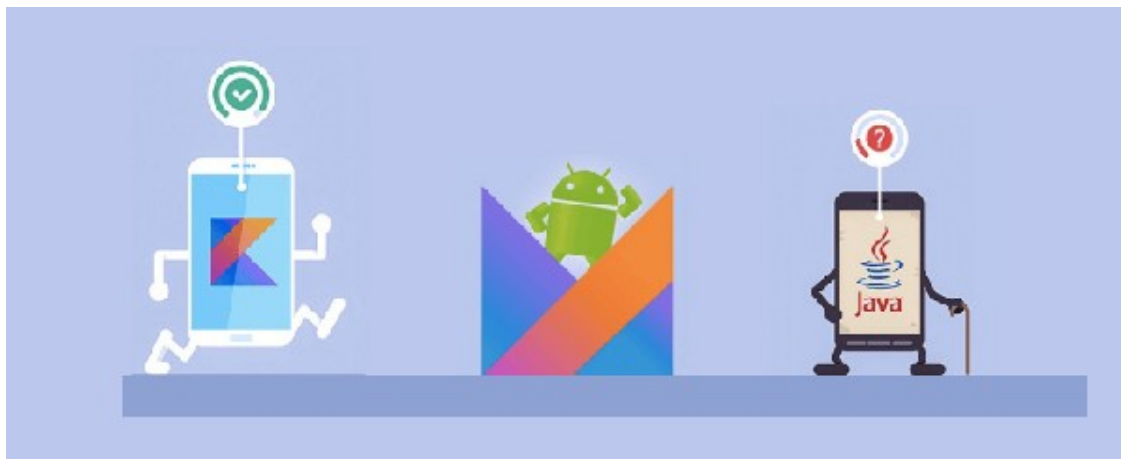
- **Rendezvous** — канал по умолчанию с ёмкостью `Channel.RENDEZVOUS (==0)`: обмен данными может произойти только, если отправитель и получатель «состыковались» (producer всегда ждёт consumer-a).
- **Conflated** — канал с ёмкостью `Channel.CONFLATED (==1)`: каждый новый элемент заменяет предыдущий.
- **Buffered** — канал с определённой ёмкостью `Channel.BUFFERED (==64 по умолчанию, изменяется в системном свойстве kotlinx.coroutines.channels.defaultBuffer в JVM)`.
- **Unlimited** — канал с ёмкостью `Channel.UNLIMITED`, то есть с неограниченным буфером, при этом функция `send` никогда не приостанавливается.



# Kotlin: применение

**Kotlin** применяется для всех видов приложений:

- мобильных (mobile: Android ← JVM; iOS, iPadOS ← native code)
- браузерных (client-side web: → JavaScript / WebAssembly)
- серверных (server-side: JVM или native?)
- настольных (desktop: MS Windows, GNU/Linux, Apple macOS)
- встраиваемых (embedded / IoT: Raspberry Pi OS @ ARM, ...)





# Kotlin Multiplatform

**Kotlin** – для кросс-платформенной разработки:

- **JVM**: server-side, desktop
- **KMM** (Kotlin Multiplatform Mobile): Android, iOS, iPadOS
- **JS**: client-side web applications
- **Native**: Windows, Linux, iOS, iPadOS, macOS, Raspberry Pi OS (ARM)



JVM



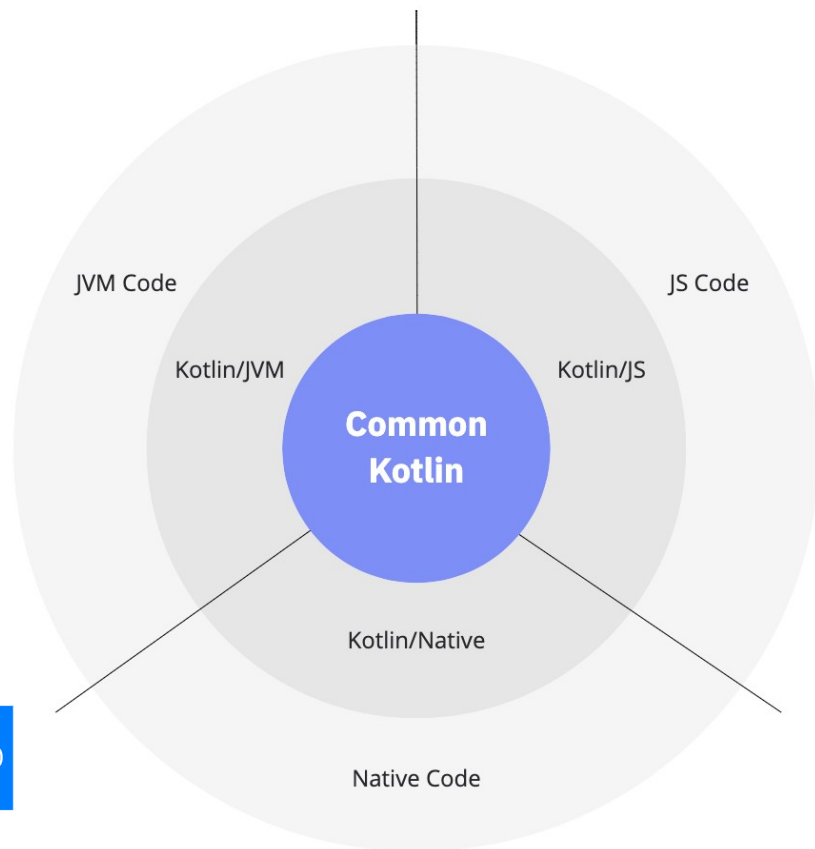
Android



Browser



Native





# Compose Multiplatform



Google **Jetpack Compose** для Android — современное средство для декларативного (без XML и шаблонов) создания UI на языке Kotlin: библиотеки `@Composable`-функций, с помощью которых представляются части визуального интерфейса.

**Compose Multiplatform** — фреймворк на Kotlin для кросс-платформенной разработки UI на основе Jetpack Compose, позволяющий использовать общие исходники:

- **Compose for Desktop** обеспечивает декларативный и реактивный подход к разработке пользовательских интерфейсов; нацелен на JVM, использует мощную графическую библиотеку Skia для поддержки высокопроизводительного аппаратно ускоренного рендеринга UI на основных платформах (macOS, Windows, Linux); позволяет создавать дистрибутивы native-приложений «в один клик».
- **Compose for Web (+ Kotlin/JS)** позволяет создавать реактивные пользовательские интерфейсы для веб-приложений на Kotlin. Предлагает разные способы объявления UI. Позволяет полностью контролировать вёрстку сайта с помощью декларативного DOM API, Multiplatform Widgets.



# Kotlin/JVM

**Kotlin** на 100% совместим с **Java** по byte-code для JVM.

Поэтому его применяют для разработки новой функциональности и переработки существующей в проектах на языке **Java**.

А также для разработки на **Kotlin** новых проектов, в которых:

- требуется совместимость с кодовой базой на **Java**;
- участвуют программисты с опытом разработки на **Java**.





# Kotlin/JS

**Kotlin/JS** предоставляет возможность переноса кода Kotlin, стандартной библиотеки Kotlin и любых совместимых зависимостей в JavaScript. Текущая реализация Kotlin/JS ориентирована на ES5 (Standard ECMA-262 ECMAScript® v.5.1).

Применение Kotlin/JS:

- Front-end в веб-приложениях: код на Kotlin для управления DOM.
- Серверные и приложения: возможность использования Node.js, предоставляемая Kotlin/JS.
- Сервисы в бессерверной инфраструктуре: типобезопасный доступ к Node.js API из кода на Kotlin.
- Мультиплатформенные проекты (использование одного кода на Kotlin в full-stack веб-приложениях): одна бизнес-логика в back-end, веб-интерфейсе и мобильном приложении.
- Библиотеки для использования с JavaScript и TypeScript: преобразовывать библиотеки из кода на Kotlin.



# Kotlin/JS

```
fun main() {  
    println("Hello, world!!!")  
}
```

Kotlin code  
to  
JavaScript code

```
kotlin.kotlin.io.output.flush();  
  
if (typeof kotlin === 'undefined') {  
    throw new Error(  
        "Error loading module 'moduleId'. Its dependency 'kotlin' was not found.  
        Please, check whether 'kotlin' is loaded prior to 'moduleId'.");  
}  
  
var moduleId = function (_, Kotlin) {  
    'use strict';  
    var println = Kotlin.kotlin.io.println_s8jyv4$;  
    function main() {  
        println('Hello, world!!!');  
    }  
    _.main = main;  
    main();  
    Kotlin.defineModule('moduleId', _);  
    return _;  
}  
(typeof moduleId === 'undefined' ? {} : moduleId, kotlin);  
  
kotlin.kotlin.io.output.buffer;
```





# Kotlin/Native



**Kotlin/Native** – средство кроссплатформенной разработки программ с компиляцией в «родной» исполняемый код (с помощью LLVM) для целевой операционной системы и аппаратной платформы.

Компилятор **Kotlin/Native** разработан для macOS, Linux и MS Windows. Он доступен как средство командной строки и поставляется в составе стандартного дистрибутива Kotlin, или его можно загрузить из раздела JetBrains на GitHub.

В **Kotlin/Native** поддерживаются разные целевые системы, включая **iOS** (arm32, arm64, simulator x86\_64), tvOS (arm64, x64), watchOS (arm32/64, x32/64), **MS Windows** (mingw32, x86\_64), **GNU/Linux** (x86\_64, MIPS32, MIPSel32, linux\_x64), **macOS** (x86\_64, arm64), **Android** NDK (arm32/64, x32/64), **Raspberry Pi OS** (arm32\_hfp, arm64), STM32, WASM.



# Kotlin/Native: КОМПИЛЯЦИЯ

```
kotlinc-native program.kt -o program -target raspberrypi
```

## # При первой компиляции загружаются библиотеки для целевой платформы

Downloading native dependencies (LLVM, sysroot etc).

This is a one-time action performed only on the first run of the compiler.

....

(KonanProperties) Downloading dependency:

<https://download.jetbrains.com/kotlin/native/qemu-arm-static-5.1.0-linux-2.tar.gz> (0/2156936).

(KonanProperties) Downloading dependency:

<https://download.jetbrains.com/kotlin/native/qemu-arm-static-5.1.0-linux-2.tar.gz> (2156936/2156936). Done.

Extracting dependency:

/home/t4503/.konan/cache/qemu-arm-static-5.1.0-linux-2.tar.gz into /home/t4503/.konan/dependencies

```
program.kt → program.kexe
```



# Kotlin/wasm



**WebAssembly** (wasm) — язык программирования низкого уровня для стековой виртуальной машины, спроектированный как кросс-платформенный целевой язык компиляции с высокоуровневых языков.

*Реализация WebAssembly есть во всех основных браузерах (кроме Internet Explorer) с конца 2017 г. По состоянию на июль 2021 год 94 % установленных браузеров поддерживают WebAssembly. Для устаревших браузеров существует полифил asm.js.*

Компилятор **Kotlin/Native** (K2) использует технологию LLVM, которая также применяется для WebAssembly.

В режиме **Kotlin/Wasm** исходник на Kotlin компилируется в код на wasm, который можно запускать в браузере.



# Kotlin for Android

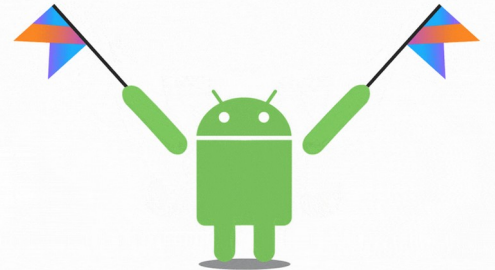
На конференции «Google I/O 2017», компания Google объявила полноценную (first-class) поддержку языка **Kotlin** на Android. Год спустя уже 35% разработчиков применяли Kotlin.

На конференции «Google I/O 2019», она объявила, что язык программирования **Kotlin** теперь стал предпочтительным языком (preferred language) для разработчиков приложений на Android.

В 2021 году 1.2 миллиона приложений в Google Play Store используют **Kotlin**.

Google оценивает, что 80% из 1000 лучших приложений в Play Store написаны на **Kotlin**. Также на нём сделаны Maps, Play, Drive.

*После установки Android Studio при создании проекта один из первых «советов дня»: «Вы можете легко преобразовать любую программу на Java в программу на Kotlin».*





# IDE для Kotlin



**IntelliJ IDEA** – IDE для разработки на Kotlin и Java,  
а также на других инструментах:

Groovy, Scala, *Ruby & JRuby*, Rust, Go, Dart,  
Python & Jython & *Cython*, *PHP*, *SQL*,  
*JavaScript & TypeScript*,

HTML & XML & JSON & YAML, XSL & Xpath, Markdown,  
*CSS & Sass & SCSS & Less*, *Haml & Slim & Liquid*.

Она также поддерживает много каркасов (frameworks)  
для разработчиков backend и frontend.

Варианты:

- IC = IDEA Community Edition.
- IE = IDEA for Education: Community Edition for learners and educators.
- IU = IDEA Ultimate Edition (*commercial, free 30-day trial*).

**Android Studio** — это специальная версия IDEA.

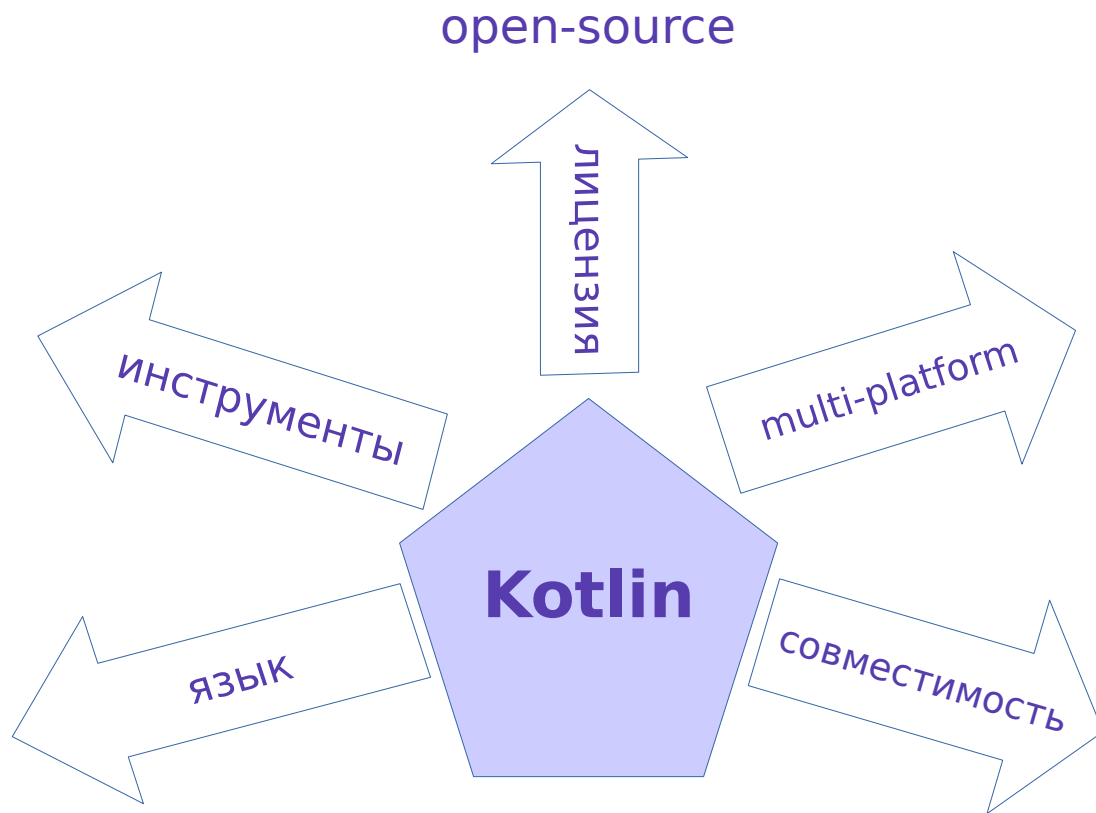




# Kotlin: достоинства

IntelliJ IDEA,  
Android Studio,  
компиляторы,  
Gradle,  
библиотеки,

современный,  
удобный,  
мощный,  
практичный,  
краткий,  
надёжный



MS Windows,  
macOS,  
GNU/Linux,  
Android,  
iOS, iPadOS,  
native@x32/64,  
native@arm32/64,  
→ JS | WASM

где *Java*,  
вместе с *Java*,  
вместо *Java*



# Kotlin стоит применять

**Kotlin** отлично подходит для обучения: особенно объектно-ориентированному и функциональному стилям программирования.

**Kotlin** очень хорош для разработки: лёгкий современный синтаксис, мощные конструкции, строгая типизация, удобные инструменты, интеграция с другими средствами разработки, море библиотек, в т. ч. из языка Java.

**Kotlin** весьма удобен для сопровождения и поддержки: короткие, ясные для понимания программы, отличная документация.

**Kotlin** эффективен в эксплуатации: многоплатформенный, программы быстро выполняются, хорошо реализована многопоточность, защита от NPE.



# Книги: много, и на русском тоже







# Ссылки: обучение

- <https://kotlinlang.org/> # Официальный сайт
- <https://kotlinlang.ru/> # ... на русском
- <https://github.com/JetBrains/kotlin> # @ GitHub
- <https://play.kotlinlang.org/> # Kotlin online playground
- <https://www.jetbrains.com/idea/download/> # IntelliJ Idea IDE
- 14 бесплатных курсов по языку программирования Kotlin
- Kotlin Basics # JetBrains Academy free course
- Введение в Kotlin JVM # @ stepik.org (курс не закончен)
- Kotlin for Java Developers # курс JetBrains @ Coursera
- Разработка Android-приложений на Kotlin # СПбГЭТУ «ЛЭТИ» + JetBrains
- Android-разработка для начинающих. Практика # @ GeekBrains
- <https://www.w3schools.com/kotlin/>
- <https://skillbox.ru/media/code/yazyk-programmirovaniya-kotlin/> # применение языка
- <https://proglib.io/p/top-10-knig-dlya-izucheniya-yazyka-programmirovaniya-kotlin>



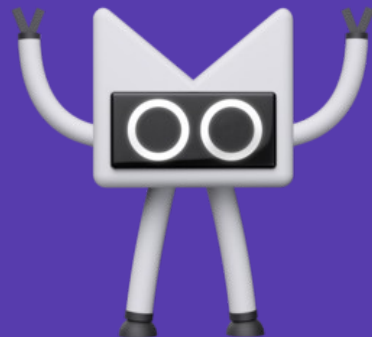
## Ссылки: дополнительно

- <https://ru.education-wiki.com/2741401-kotlin-functions> # Функции в Котлине
- <https://www.youtube.com/watch?v=aR1LluibAD0> # Введение в Kotlin Coroutines
- <https://kt.academy/article/cc-channel> # Каналы между сопрограммами
- <https://github.com/Kotlin/dokka> # API documentation engine for Kotlin
- <https://superkotlin.com/kotlin-and-webassembly/> # Kotlin and WebAssembly
- <https://blog.jdriven.com/2021/04/running-kotlin-in-the-browser-with-wasm/>
- <https://metanit.com/kotlin/jetpack/> # Руководство созданию приложений под Android с помощью Kotlin и Jetpack Compose



**Готов ответить на ваши**

**? ? ?**



Посвящается нашему сыну Лёве.

