Kotlin Exam 2025

1. Точки входа в программу. Переменные только для чтения. Выведение типа переменной. Изменяемые переменные. Переменные (свойства) верхнего уровня.

这是 Kotlin 语言的基础语法部分。虽然之前的题目是 Java，但这几个术语（尤其是“顶层属性”和 `val/var` 的区分）明显是 Kotlin 的特征。
我们将这道题拆解为 5 个小点来回答。

1. Точки входа в программу

程序的入口点
在 Kotlin 中，程序从 `main` 函数开始。与 Java 不同，它不需要放在类里面。

```kotlin
// RU: Начиная с Kotlin 1.3, параметры args не обязательны.
// EN: Since Kotlin 1.3, args parameters are optional.
// CN: 从 Kotlin 1.3 开始，args 参数是可选的。

fun main() {
    println("Hello, Kotlin!")
}

// RU: Старый стиль (как в Java).
// CN: 旧风格（像 Java 一样）。
// fun main(args: Array<String>) { ... }
```

- Объяснение (解释): Точка входа — это функция `main`, определенная на верхнем уровне (вне класса). Это аналог `public static void main` в Java, но с более лаконичным синтаксисом. 入口点是定义在顶层（类之外）的 `main` 函数。它是 Java 中 `public static void main` 的等价物，但语法更简洁。

2. Переменные только для чтения (val)

只读变量 (val)
这是 Kotlin 推荐的默认方式。相当于 Java 的 `final` 变量。

```kotlin
// RU: Ключевое слово 'val' (Value).
// EN: Keyword 'val' (Value).
// CN: 关键字 'val' (Value)。
val name = "Gemini"

// RU: Ошибка компиляции! Переназначить нельзя.
// EN: Compilation Error! Cannot reassign.
// CN: 编译错误！不能重新赋值。
// name = "GPT"
```

- Объяснение (解释): Переменные, объявленные через `val`, являются неизменяемыми (immutable reference). Их значение присваивается один раз. 使用 `val` 声明的变量是不可变的（引用不可变）。它们的值只能被赋值一次。

3. Выведение типа переменной (Type Inference)

变量类型推断
Kotlin 编译器非常聪明，通常不需要显式写出类型。

```kotlin
// RU: Мы не пишем ": String", компилятор сам это понял.
// EN: We don't write ": String", the compiler understood it itself.
// CN: 我们没写 ": String"，编译器自己看出来了。
val message = "Hello"

// RU: Здесь тип выводится как Int.
// CN: 这里类型被推断为 Int。
val count = 42

// RU: Явное указание типа (если очень нужно).
// CN: 显式指定类型（如果非常需要）。
val explicitDouble: Double = 3.14
```

- Объяснение (解释): Компилятор анализирует правую часть выражения и автоматически определяет тип переменной. Это уменьшает количество шаблонного кода. 编译器分析表达式的右侧，并自动确定变量的类型。这减少了样板代码的数量。

4. Изменяемые переменные (var)

可变变量 (var)
如果你需要在程序运行过程中改变变量的值，使用 `var`。

```kotlin
// RU: Ключевое слово 'var' (Variable).
// EN: Keyword 'var' (Variable).
// CN: 关键字 'var' (Variable)。
var score = 10

// RU: Теперь можно менять значение.
// EN: Now we can change the value.
// CN: 现在可以改变值了。
score = 20
score = score + 5
```

- Объяснение (解释): Переменные `var` могут быть переназначены. Однако их тип изменить нельзя (если она была Int, она останется Int). `var` 变量可以被重新赋值。但是，它们的类型不能改变（如果它是 Int，它就永远是 Int）。

5. Переменные (свойства) верхнего уровня

顶层变量（属性）
在 Java 中，一切都必须在类里。在 Kotlin 中，变量可以定义在文件级别（类之外）。

```kotlin
package com.example.app

// RU: Эта переменная видна во всем файле (и проекте).
// EN: This variable is visible in the whole file (and project).
// CN: 这个变量在整个文件（及项目）中都可见。
val APP_VERSION = "1.0.0"
var clickCount = 0

class User {
    fun printInfo() {
        // RU: Можно использовать переменную верхнего уровня внутри класса.
        // CN: 可以在类内部使用顶层变量。
        println("Version: $APP_VERSION")
    }
}
```

- Объяснение (解释): Свойства верхнего уровня объявляются вне какого-либо класса. Они заменяют статические поля (`static fields`) из Java и доступны из любой части кода (в зависимости от модификаторов видимости). 顶层属性是在任何类之外声明的。它们取代了 Java 中的静态字段（`static fields`），并且可以从代码的任何部分访问（取决于可见性修饰符）。

2. Хункции. Вывод типа значения функции. Умалчиваемые аргументы функции. Именованные аргументы функций. Хункции с переменным числом параметров. Хункции-расширения. Свойства-расширения для чтения и записи. Инфиксные функции. Хункции-операторы.

Это **Kotlin** (часть 2), посвященная функциям. В Kotlin функции — это граждане первого класса, и они намного гибче, чем методы в Java.
Мы разберем каждый пункт вашего вопроса с примерами кода и пояснениями на русском и китайском языках.

1. Хункции (Functions) и Вывод типа значения (Type Inference)

函数与返回值类型推断
В Kotlin функции объявляются с помощью ключевого слова `fun`.
Стандартный вид (Standard):

```kotlin
// Блочное тело (Block Body): Тип возвращаемого значения обязателен (если не
        Unit).
// 块体：必须显式声明返回值类型（除非是 Unit）。
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

Краткий вид с выводом типа (Single-Expression): Если функция состоит из одного выражения, можно опустить фигурные скобки и тип возвращаемого значения.

```kotlin
// RU: Компилятор сам понимает, что (Int + Int) это Int.
// CN: 编译器自己明白 (Int + Int) 的结果是 Int。
fun sumShort(a: Int, b: Int) = a + b
```

2. Умалчиваемые аргументы (Default Arguments)

默认参数
Больше не нужно писать 10 перегруженных методов (Overloading), как в Java! Можно просто задать значение по умолчанию.

```kotlin
// RU: Если параметр 'message' не передан, используется "Default".
// CN: 如果没传 'message' 参数，就使用 "Default"。
fun printLog(tag: String, message: String = "Default") {
    println("$tag: $message")
}

// Использование:
printLog("System") // Output: System: Default
printLog("User", "Hello") // Output: User: Hello
```

3. Именованные аргументы (Named Arguments)

命名参数
Вы можете передавать аргументы в любом порядке, если указываете их имена. Это делает код очень читаемым.

```kotlin
fun formatText(text: String, normalize: Boolean, upperCase: Boolean) { ... }

// RU: Понятно, что значит каждый boolean. Порядок не важен.
// CN: 每个 boolean 的含义一目了然。顺序不再重要。
formatText(
    text = "Hello",
    upperCase = true,
    normalize = false
)
```

4. Хункции с переменным числом параметров (Varargs)

可变参数函数
Используется ключевое слово `vararg` (аналог `...` в Java).

```kotlin
// RU: Принимает любое количество чисел.
// CN: 接收任意数量的数字。
fun printNumbers(vararg numbers: Int) {
    for (n in numbers) {
        print("$n ")
    }
}

// Использование:
printNumbers(1, 2, 3, 4, 5)

// Spread operator (*):
val array = intArrayOf(10, 20)
// RU: Чтобы передать массив в vararg, нужно использовать звездочку (*).
// CN: 要把数组传给 vararg，必须使用星号 (*)。
printNumbers(*array)
```

5. Хункции-расширения (Extension Functions)

扩展函数

Это “магия” Kotlin. Вы можете добавить новый метод в существующий класс (даже в `String` или `List`), не наследуясь от него.

```
// RU: Добавляем метод sayHello в класс String.
// CN: 给 String 类添加 sayHello 方法。
fun String.sayHello(): String {
    // 'this' ссылается на саму строку.
    // 'this' 指的是字符串本身。
    return "Hello, $this!"
}

val name = "World"
println(name.sayHello()) // Output: Hello, World!
```

6. Свойства-расширения (Extension Properties)

扩展属性

Как и функции, но для свойств. Можно добавить только геттер/сеттер. Поля (памяти) у них нет.

```
// RU: Добавляем свойство lastChar в String.
// CN: 给 String 添加 lastChar 属性。
val String.lastChar: Char
    get() = this.get(this.length - 1)

// Использование:
println("Kotlin".lastChar) // Output: n
```

- Ограничение (Limitation): Свойства-расширения не могут иметь `backing field` (не могут хранить состояние), они вычисляются при каждом обращении.
- 限制： 扩展属性不能有 `backing field` （不能存储状态），每次访问都会重新计算。

7. Инфиксные функции (Infix Functions)

中缀函数

Позволяют вызывать функцию без точки и скобок. Делает код похожим на естественный язык.

Требования:

1. Модификатор `infix`.
2. Хункция-член или расширение.
3. Ровно один параметр.

```
infix fun Int.add(x: Int): Int {
    return this + x
}

// Использование:
// Обычный вызов (Standard call)
val a = 1.add(2)

// Инфиксный вызов (Infix call)
// RU: Выглядит как оператор!
// CN: 看起来像操作符！
val b = 1 add 2
```

8. Хункции-операторы (Operator Functions)

操作符重载函数

Kotlin позволяет перегружать стандартные операторы (`+`, `-`, `*`, `[]`, `==`) с помощью модификатора `operator`.

```
data class Point(val x: Int, val y: Int) {
    // RU: Перегружаем оператор '+' (plus).
    // CN: 重载 '+' (plus) 操作符。
    operator fun plus(other: Point): Point {
        return Point(x + other.x, y + other.y)
    }
}

val p1 = Point(10, 20)
val p2 = Point(5, 5)

// RU: Теперь мы можем складывать объекты через +
// CN: 现在我们可以通过 + 来相加对象
val p3 = p1 + p2 // Point(15, 25)
```

Краткий словарь операторов (Map):

- `plus` -> `+`
- `minus` -> `-`
- `times` -> `*`
- `div` -> `/`
- `get(i)` -> `a[i]`
- `invoke()` -> `a()`

3. Пакеты. Импорты. КЛАССЫ. Классы данных. Классы-перечисления. Запечатанные классы. Объекты “Одиночки”, Объекты-компаньены, Объекты-выражения. Интерфейсы.

Это продолжение темы **Kotlin**, посвященное объектно-ориентированному программированию (ООП) и структуре кода. Kotlin значительно упрощает создание классов по сравнению с Java. Мы разберем каждый пункт вашего запроса.

1. Пакеты и Импорты (Packages & Imports)

包与导入

В Kotlin, как и в Java, код организуется в пакеты. Однако, в отличие от Java, структура пакетов не обязана совпадать со структурой папок.

```
// В начале файла (At the top of the file)
package com.example.myapp

// Импорт конкретного класса
// 导入特定类
import java.util.Random

// Импорт всего содержимого пакета (как *)
// 导入包的所有内容
import java.util.*

// RU: Можно переименовать класс при импорте (устранение конфликтов имен).
// CN: 导入时可以重命名类（解决命名冲突）。
import java.sql.Date as SqlDate
```

2. Классы (Classes)

类

Классы в Kotlin объявляются ключевым словом `class`. По умолчанию все классы — **final** (от них нельзя наследоваться). Чтобы разрешить наследование, нужно использовать `open`.

```
// RU: Первичный конструктор объявляется прямо в заголовке класса.
// CN: 主构造器直接在类头中声明。
class Person(val name: String, var age: Int) {

    // RU: Блок инициализации (выполняется при создании).
    // CN: 初始化块（创建时执行）。
    init {
        println("Created person: $name")
    }

    fun sayHello() {
        println("Hi, I am $name")
    }
}

// Создание объекта (без слова new!):
// 创建对象（不需要 new 关键字！）：
val p = Person("Alex", 25)
```

3. Классы данных (Data Classes)

数据类

Это аналог Java 16 `record` или Lombok `@Data`. Используются для хранения данных.

Kotlin автоматически генерирует:

- `toString()`
- `equals()` / `hashCode()`
- `copy()`
- `componentN()` (для деструктуризации)

```
// Ключевое слово 'data'
data class User(val name: String, val id: Int)

fun main() {
    val u1 = User("Alice", 1)
    val u2 = User("Alice", 1)

    // RU: Автоматический toString()
    println(u1) // User(name=Alice, id=1)

    // RU: Сравнение по содержимому (equals), а не по ссылкам.
    // CN: 按内容比较 (equals)，而不是按引用。
    println(u1 == u2) // true

    // RU: Метод copy() для создания копии с изменением полей.
    // CN: copy() 方法用于创建副本并修改字段。
    val u3 = u1.copy(id = 2)
}
```

4. Классы-перечисления (Enum Classes)

枚举类

Почти как в Java, но ключевых слова два: `enum class`.

```
enum class Status(val code: Int) {
    OPEN(100),
    IN_PROGRESS(102),
    CLOSED(200); // Точка с запятой обязательна, если дальше идут методы.

    fun isFinished(): Boolean = this == CLOSED
}
```

5. Запечатанные классы (Sealed Classes)

密封类

Это “супер-энумы”. Они ограничивают иерархию наследования. Все наследники должны быть известны компилятору. Идеально подходят для обработки состояний (State) в архитектуре (например, MVI/MVVM).

```
// RU: Наследоваться могут только классы в этом же файле/пакете.
// CN: 只有同一个文件/包中的类可以继承它。
sealed class Result

// Наследники могут быть data class или object
data class Success(val data: String) : Result()
data class Error(val exception: Exception) : Result()
object Loading : Result()

fun handleResult(result: Result) {
    // RU: В 'when' не нужна ветка 'else', так как компилятор знает все
    варианты.
    // CN: 'when' 中不需要 'else' 分支，因为编译器知道所有选项。
    when (result) {
        is Success -> println(result.data)
        is Error -> println(result.exception)
        Loading -> println("Loading...")
    }
}
```

6. Объекты (Objects)

В Kotlin ключевое слово `object` используется в трех разных ситуациях.

A. Объекты “Одиночки” (Object Declaration)

单例对象 (Singleton) Это реализация паттерна Singleton “из коробки”.

```kotlin
// RU: Этот объект существует в единственном экземпляре.
// CN: 这个对象只有一个实例。
object DatabaseConfig {
    val url = "jdbc:mysql://localhost"
    fun connect() { /*...*/ }
}

// Использование:
DatabaseConfig.connect()
```

B. Объекты-компаньоны (Companion Objects)

伴生对象 В Kotlin нет ключевого слова `static`. Вместо статических методов используется `companion object`.

```kotlin
class MyClass {
    // Обычный метод
    fun method() {}

    // RU: Компаньон. Его методы вызываются через имя класса.
    // CN: 伴生对象。其方法通过类名调用。
    companion object {
        fun staticMethod() {
            println("I am like static")
        }
    }
}

// Вызов:
MyClass.staticMethod()
```

C. Объекты-выражения (Object Expressions)

对象表达式 (匿名类) Замена анонимным классам в Java. Используется, когда нужно создать объект интерфейса "на лету".

```kotlin
interface ClickListener {
    fun onClick()
}

fun setListener(listener: ClickListener) {}

fun main() {
    // RU: Создаем анонимный объект.
    // CN: 创建匿名对象。
    setListener(object : ClickListener {
        override fun onClick() {
            println("Clicked!")
        }
    })
}
```

7. Интерфейсы (Interfaces)

接口

Интерфейсы в Kotlin могут содержать не только абстрактные методы, но и реализацию по умолчанию, а также абстрактные свойства.

```kotlin
interface Movable {
    // Абстрактное свойство (наследник должен переопределить геттер)
    val speed: Int

    // Абстрактный метод
    fun move()

    // Метод с реализацией по умолчанию
    // 带有默认实现的方法
    fun stop() {
        println("Stopped")
    }
}

class Car(override val speed: Int) : Movable {
    override fun move() {
        println("Moving at $speed km/h")
    }
}
```

4. Циклы, Оператор in. Диапазоны (ranges). Использование числовых диапазонов в циклах. Использование диапазонов в условном выражении. Выбирающее предложение when. Условное выражение if. ИСКЛЮЧЕНИЯ. Порождение исключения. Перехват исключения. Исключения как выражения.

Это продолжение темы **Kotlin**, посвященное управлению потоком выполнения (Control Flow) и обработке ошибок. 这是 **Kotlin** 的延续，主要讲解控制流 (Control Flow) 和错误处理。
Мы разберем каждый пункт строго по порядку. 我们将严格按照顺序解析每一点。

1. Циклы (Loops)

循环

В Kotlin основными циклами являются `for`, `while` и `do-while`. Kotlin 中的主要循环是 `for`、`while` 和 `do-while`。

```kotlin
// 1. Цикл for (перебор элементов)
// 1. for 循环（遍历元素）
val list = listOf("A", "B", "C")
for (item in list) {
    println(item)
}

// 2. Цикл while (пока условие истинно)
// 2. while 循环（当条件为真时）
var x = 5
while (x > 0) {
    x--
}
```

2. Оператор in

in 操作符

Оператор `in` используется для проверки вхождения элемента в коллекцию или диапазон, а также для итерации. `in` 操作符用于检查元素是否属于集合或范围，也用于迭代。

```kotlin
val names = listOf("Ivan", "Maria")

// RU: Проверка: есть ли "Ivan" в списке?
// CN: 检查："Ivan" 在列表中吗？
if ("Ivan" in names) {
    println("Found!")
}

// RU: Итерация: для каждого элемента в списке.
// CN: 迭代：对于列表中的每个元素。
for (name in names) { /* ... */ }
```

3. Диапазоны (Ranges)

范围

Диапазоны позволяют легко создавать последовательности чисел. 范围允许轻松创建数字序列。

```kotlin
// RU: Диапазон от 1 до 10 (включительно).
// CN: 从 1 到 10 的范围（包含 10）。
val oneToTen = 1..10

// RU: Диапазон от 1 до 9 (10 не включается).
// CN: 从 1 到 9 的范围（不包含 10）。
val oneToNine = 1 until 10
```

4. Использование числовых диапазонов в циклах

在循环中使用数字范围

Мы можем использовать диапазоны напрямую в цикле `for`. 我们可以直接在 `for` 循环中使用范围。

```kotlin
// RU: Простой перебор: 1, 2, 3, 4, 5.
// CN: 简单遍历：1, 2, 3, 4, 5。
for (i in 1..5) {
    println(i)
}

// RU: Обратный отсчет: 5, 4, 3, 2, 1.
// CN: 倒数：5, 4, 3, 2, 1。
for (i in 5 downTo 1) {
    print(i)
}

// RU: С шагом: 1, 3, 5, 7...
// CN: 带步长：1, 3, 5, 7...
for (i in 1..10 step 2) {
    print(i)
}
```

5. Использование диапазонов в условном выражении

在条件表达式中使用范围

Очень удобно проверять, попадает ли число в заданный интервал. 检查数字是否落在给定区间内非常方便。

```kotlin
val age = 25

// RU: Если возраст от 18 до 60 (включительно).
// CN: 如果年龄在 18 到 60 岁之间（含）。
if (age in 18..60) {
    println("Worker")
}

// RU: Проверка "НЕ входит".
// CN: 检查 "不属于"。
if (age !in 0..100) {
    println("Invalid age")
}
```

6. Выбирающее предложение when

when 选择语句

`when` — это мощная замена оператору `switch` из Java/C++. Он может возвращать значение. `when` 是 Java/C++ 中 `switch` 操作符的强大替代品。它可以返回值。

```kotlin
val x = 10

// RU: when как выражение (возвращает String).
// CN: when 作为表达式（返回 String）。
val result = when (x) {
    1 -> "One"
    2, 3 -> "Two or Three"
    in 10..20 -> "Between 10 and 20"
    else -> "Unknown"
}
```

7. Условное выражение if

if 条件表达式

В Kotlin `if` — это выражение (expression), оно возвращает значение. Тернарного оператора (`? :`) в Kotlin нет, потому что `if` выполняет его роль. 在 Kotlin 中，`if` 是一个表达式 (expression)，它会返回值。Kotlin 中没有三元运算符 (`? :`)，因为 `if` 扮演了这个角色。

```kotlin
val a = 10
val b = 20

// RU: Вместо: int max = (a > b) ? a : b;
```

```kotlin
// CN: 代替：int max = (a > b) ? a : b;
val max = if (a > b) a else b

// RU: Можно использовать блоки кода (возвращается последняя строка).
// CN: 可以使用代码块（返回最后一行）。
val min = if (a < b) {
    println("a is smaller")
    a // Return value
} else {
    b // Return value
}
```

8. ИСКЛЮЧЕНИЯ (Exceptions)

异常

Все классы исключений в Kotlin наследуются от `Throwable`. В Kotlin нет "проверяемых исключений" (Checked Exceptions). Вам не нужно писать `throws IOException`. Kotlin 中的所有异常类都继承自 `Throwable`。Kotlin 中没有"受检异常" (Checked Exceptions)。你不需要写 `throws IOException`。

9. Порождение исключения

抛出异常

Используется ключевое слово `throw`. 使用 `throw` 关键字。

```kotlin
fun validate(age: Int) {
    if (age < 0) {
        // RU: Бросаем исключение.
        // CN: 抛出异常。
        throw IllegalArgumentException("Age cannot be negative")
    }
}
```

10. Перехват исключения

捕获异常

Используется стандартная конструкция `try-catch-finally`. 使用标准的 `try-catch-finally` 结构。

```kotlin
try {
    // Code that may fail
    validate(-5)
} catch (e: IllegalArgumentException) {
    // RU: Обработка ошибки.
    // CN: 错误处理。
    println("Error: ${e.message}")
} finally {
    // RU: Выполняется всегда.
    // CN: 总是执行。
    println("Done")
}
```

11. Исключения как выражения

异常作为表达式

В Kotlin `try` — это тоже выражение! Оно может возвращать результат. 在 Kotlin 中，`try` 也是一个表达式！它可以返回结果。

```kotlin
val input = "123a"

// RU: Пытаемся распарсить число. Если ошибка — возвращаем null (или 0).
// CN: 尝试解析数字。如果出错 — 返回 null（或 0）。
val number: Int? = try {
    input.toInt()
} catch (e: NumberFormatException) {
    null // RU: Это значение пойдет в переменную number. (CN: 这个值将赋给变量 number。)
}

println(number) // Output: null
```

5. Перегрузка операторов. Унарные префиксные операторы. Арифметические операции. Операторы сравнений. Оператор in. Оператор доступа по индексу. Оператор вызова. Присвоения с накоплением.

Это 5-й вопрос (Question 5) из вашего списка по Kotlin. 这是您列表中的 第 5 题 (Question 5)，关于 Kotlin 的操作符重载。

В Kotlin перегрузка операторов — это возможность задать свою реализацию для стандартных знаков (`+`, `-`, `*`, `/` и т.д.) с помощью ключевого слова `operator` и специальных имен функций. 在 Kotlin 中，操作符重载允许你通过关键字 `operator` 和特定的函数名为标准符号（`+`, `-`, `*`, `/` 等）定义自己的实现。

Мы разберем каждый пункт подробно. 我们将逐一详细解析。

1. Перегрузка операторов (Operator Overloading)

操作符重载

Для перегрузки оператора нужно объявить функцию с ключевым словом `operator` и зарезервированным именем (например, `plus` для `+`). 要重载操作符，需要声明一个带有 `operator` 关键字和保留名称（例如 `plus` 对应 `+`）的函数。

```kotlin
data class Point(val x: Int, val y: Int) {
    // RU: Объявляем оператор.
    // CN: 声明操作符。
    operator fun plus(other: Point): Point {
        return Point(x + other.x, y + other.y)
    }
}
```

2. Унарные префиксные операторы

一元前缀操作符

Эти операторы применяются к одной переменной (`-a`, `!a`, `++a`). 这些操作符作用于单个变量 (`-a`, `!a`, `++a`)。

Таблица имен (Name Table):

- `+a` -> `unaryPlus()`
- `-a` -> `unaryMinus()`
- `!a` -> `not()`
- `++a` -> `inc()` (Increment / 自增)
- `--a` -> `dec()` (Decrement / 自减)

```kotlin
data class Counter(val value: Int) {
    // RU: Перегрузка унарного минуса (-a).
    // CN: 重载一元负号 (-a)。
    operator fun unaryMinus(): Counter {
        return Counter(-value)
    }
}

val c = Counter(10)
val negative = -c // Counter(-10)
```

3. Арифметические операции

算术运算

Это стандартные бинарные операции (`a + b`). 这是标准的二元运算 (`a + b`)。

Таблица имен (Name Table):

- `a + b` -> `plus`
- `a - b` -> `minus`
- `a * b` -> `times`
- `a / b` -> `div`
- `a % b` -> `rem` (остаток / 取余)

```kotlin
// RU: Пример умножения (times).
// CN: 乘法示例 (times)。
operator fun times(scale: Int): Point {
    return Point(x * scale, y * scale)
}
```

4. Операторы сравнений

比较操作符

В Kotlin операторы сравнения (`>`, `<`, `>=`, `<=`) преобразуются в вызов функции `compareTo`. 在 Kotlin 中，比较操作符 (`>`, `<`, `>=`, `<=`) 会被转换为对 `compareTo` 函数的调用。

Правило (Rule):

- `a > b` превращается в `a.compareTo(b) > 0`
- `a < b` превращается в `a.compareTo(b) < 0`

```kotlin
class Person(val age: Int) : Comparable<Person> {
    // RU: Функция должна возвращать Int (отрицательное, ноль или положительное число).
    // CN: 该函数必须返回 Int（负数、零或正数）。
    override fun compareTo(other: Person): Int {
        return this.age - other.age
    }
}

val p1 = Person(20)
val p2 = Person(30)
val result = p1 < p2 // true
```

- Примечание: `==` преобразуется в `equals()`, но `equals` не требует ключевого слова `operator` (это переопределение из `Any`).
- 注意： `==` 转换为 `equals()`，但 `equals` 不需要 `operator` 关键字（它是对 `Any` 的重写）。

5. Оператор in

`in` 操作符

Используется для проверки вхождения элемента в коллекцию (`a in b`). Преобразуется в функцию `contains`. 用于检查元素是否包含在集合中 (`a in b`)。转换为 `contains` 函数。

Таблица имен (Name Table):

- `a in b` -> `b.contains(a)`
- `a !in b` -> `!b.contains(a)`

```kotlin
data class Rectangle(val left: Int, val right: Int) {
    // RU: Проверяем, находится ли точка внутри границ.
    // CN: 检查点是否在边界内。
    operator fun contains(x: Int): Boolean {
        return x in left..right
    }
}

val rect = Rectangle(0, 10)
// RU: Вызовется rect.contains(5).
// CN: 将调用 rect.contains(5)。
val isInside = 5 in rect // true
```

6. Оператор доступа по индексу

索引访问操作符

Позволяет обращаться к объекту как к массиву или списку через квадратные скобки `[]`. 允许像数组或列表一样通过方括号 `[]` 访问对象。

Таблица имен (Name Table):

- `a[i]` (чтение / 读) -> `get(i)`
- `a[i] = b` (запись / 写) -> `set(i, b)`

```kotlin
class Matrix {
    // RU: Чтение a[i, j].
    // CN: 读取 a[i, j]。
    operator fun get(row: Int, col: Int): Int { return 0 }

    // RU: Запись a[i, j] = value.
    // CN: 写入 a[i, j] = value。
    operator fun set(row: Int, col: Int, value: Int) { /*...*/ }
}

val m = Matrix()
m[1, 2] = 5 // set(1, 2, 5)
val x = m[1, 2] // get(1, 2)
```

7. Оператор вызова (Invoke)

调用操作符

Позволяет вызывать объект, как будто это функция. 允许像调用函数一样调用对象。
Таблица имен (Name Table):
- `a()` -> `a.invoke()`
- `a(i)` -> `a.invoke(i)`

```
class Greeter {
    // RU: Оператор invoke.
    // CN: invoke 操作符。
    operator fun invoke(name: String) {
        println("Hello, $name!")
    }
}

val greet = Greeter()
// RU: Вызываем объект как функцию!
// CN: 像函数一样调用对象！
greet("Kotlin")
```

8. Присвоения с накоплением (Augmented Assignments)
累加赋值 (复合赋值)
Это операторы типа `+=`, `-=`, `*=`. 这是 `+=`, `-=`, `*=` 类型的操作符。
Логика (Logic):
1. Если определена функция `plusAssign`, используется она (изменяет объект, `void`).
2. Если нет, используется `plus` (создает новый объект: `a = a + b`).
3. 如果定义了 `plusAssign` 函数，则使用它（修改对象，返回 `void`）。
4. 如果没有，则使用 `plus`（创建新对象：`a = a + b`）。

```
data class Box(var items: Int) {
    // RU: Изменяем текущий объект.
    // CN: 修改当前对象。
    operator fun plusAssign(other: Int) {
        this.items += other
    }
}

val box = Box(5)
box += 3 // items = 8
```

6. Объявление интерфейсов. Наследование интерфейсов. Реализация интерфейсов. Реализация свойств интерфейсов.

7. Модификаторы класса abstract open final sealed. Модификаторы видимости public, internal, protected, private. Основной и вторичные конструкторы класса. Свойства класса, реализация свойств интерфейсов. Свойства с поздней инициализацией. Внутренние и вложенные классы. Универсальные методы классов.

Это 6-й вопрос (Question 6), посвященный интерфейсам в Kotlin. 这是 第 6 题 (Question 6)，关于 Kotlin 中的接口。
В Kotlin интерфейсы очень похожи на Java 8+, так как они могут содержать реализацию методов по умолчанию и абстрактные свойства. 在 Kotlin 中，接口非常类似于 Java 8+，因为它们可以包含默认方法实现和抽象属性。

1. Объявление интерфейсов
接口声明
Для создания интерфейса используется ключевое слово `interface`. Методы могут быть абстрактными (без тела) или иметь реализацию по умолчанию. 使用关键字 `interface` 创建接口。方法可以是抽象的（没有方法体），也可以有默认实现。

```
// RU: Объявление интерфейса.
// CN: 声明接口。
interface Clickable {
    // RU: Абстрактный метод (нужно реализовать).
    // CN: 抽象方法（必须实现）。
    fun click()

    // RU: Метод с реализацией по умолчанию (не обязательно переопределять).
    // CN: 带有默认实现的方法（不强制重写）。
    fun show() {
        println("I am clickable!")
    }
}
```

2. Наследование интерфейсов
接口继承
Один интерфейс может наследовать другой. В этом случае наследуемый интерфейс может переопределить методы родителя или оставить их как есть. 一个接口可以继承另一个接口。在这种情况下，继承的接口可以重写父类的方法，也可以保持原样。

```
// RU: Интерфейс Resizable наследует Clickable.
// CN: 接口 Resizable 继承 Clickable。
interface Resizable : Clickable {
    fun resize()
}
```

3. Реализация интерфейсов
接口实现
Класс или объект реализует интерфейс, используя двоеточие `:`. Все абстрактные члены должны быть переопределены с помощью `override`. 类或对象使用冒号 `:` 实现接口。必须使用 `override` 重写所有抽象成员。

```
// RU: Класс Button реализует Clickable.
// CN: Button 类实现 Clickable。
class Button : Clickable {
    // RU: Обязательная реализация.
    // CN: 必须实现。
    override fun click() {
        println("Button clicked")
    }

    // RU: Метод show() можно не переопределять, возьмется версия из интерфейса.
    // CN: show() 方法可以不重写，会使用接口中的版本。
}
```

4. Реализация свойств интерфейсов
接口属性的实现
Это уникальная особенность Kotlin. Интерфейсы могут содержать свойства, но они не могут хранить состояние (у них нет `backing field`). 这是 Kotlin 的独特功能。接口可以包含属性，但它们不能存储状态（没有 `backing field`，即幕后字段）。
Варианта два (Two options):
1. Абстрактное свойство (Abstract): Класс должен предоставить значение.
2. Свойство с геттером (With Accessor): Вычисляет значение каждый раз.

```
interface User {
    // RU: Абстрактное свойство. Класс-наследник должен хранить эти данные.
    // CN: 抽象属性。子类必须存储此数据。
    val email: String

    // RU: Свойство с геттером. Оно не занимает память, а вычисляется на лету.
    // CN: 带 getter 的属性。它不占用内存，而是即时计算。
    val nickname: String
        get() = email.substringBefore('@')
}

class PrivateUser(override val email: String) : User {
    // RU: nickname вычисляется автоматически на основе email.
    // CN: nickname 根据 email 自动计算。
}
```

8. Иерархия типов языка Kotlin. Поддержка значения null. Оператор безопасного вызова ?. Оператор Элвис ?: Проверка на null с помощью оператора !! Безопасное приведение типов: оператор as? Хункция let. Свойства с отложенной инициализацией. Расширение типов поддерживающих значение null. Типы и подтипы, классы и подклассы. Изменяемые и неизменяемые коллекции.

Это 7-й вопрос (Question 7), охватывающий систему типов Kotlin и безопасность Null (Null Safety). 这是 第 7 题 (Question 7)，涵盖了 Kotlin 的类型系统和空安全 (Null Safety)。
Мы разберем каждое предложение как отдельный вопрос. 我们将把每一句话作为一个单独的问题来解析。

1. Иерархия типов языка Kotlin
Kotlin 语言的类型层次结构
В Kotlin вершиной иерархии является класс `Any`. 在 Kotlin 中，类型层次结构的顶端是 `Any` 类。
- **Any:** Корень всех не-null типов (аналог `Object` в Java).
- **Any:** 所有非 null 类型的根（类似于 Java 中的 `Object`）。
- **Any?:** Корень вообще всех типов (включая null).
- **Any?:** 所有类型的根（包括 null）。
- **Unit:** Аналог `void` (возвращается функциями, которые ничего не возвращают).
- **Unit:** 类似于 `void`（由不返回任何内容的函数返回）。
- **Nothing:** Тип, у которого нет значений. Используется для функций, которые никогда не завершаются (например, выбрасывают исключение).
- **Nothing:** 没有值的类型。用于永远不会正常结束的函数（例如抛出异常）。

2. Поддержка значения null
空值 (null) 支持
Kotlin разделяет типы на те, которые могут быть null, и те, которые не могут. Kotlin 将类型分为可以是 null 的类型和不可以是 null 的类型。

```
// RU: Обычная строка. Null запрещен. Ошибка компиляции при присвоении null.
// CN: 普通字符串。禁止 Null。赋值 null 会导致编译错误。
var a: String = "abc"
// a = null // Error!

// RU: Строка, допускающая null (Nullable). Нужен вопросительный знак (?).
// CN: 允许 null 的字符串 (Nullable)。需要问号 (?)。
var b: String? = "abc"
b = null // OK
```

3. Оператор безопасного вызова ?. (Safe Call Operator)
安全调用操作符 ?.
Позволяет обратиться к свойству или методу, только если объект не null. Если объект null, выражение возвращает null, а не падает с ошибкой. 允许仅当对象不为 null 时访问属性或方法。如果对象为 null，表达式返回 null，而不是因错误崩溃。

```
val b: String? = null

// RU: Если b не null, вернет длину. Если b == null, вернет null.
// CN: 如果 b 不为 null，返回长度。如果 b == null，返回 null。
val len: Int? = b?.length
```

4. Оператор Элвис ?: (Elvis Operator)
Elvis 操作符 ?:
Используется для предоставления значения по умолчанию, если выражение слева равно null. 用于在左侧表达式为 null 时提供默认值。

```
val b: String? = null

// RU: Если длина есть — берем её. Если null — берем -1.
```

```kotlin
// CN: 如果有长度则取长度。如果是 null，则取 -1。
val len: Int = b?.length ?: -1
```

5. Проверка на null с помощью оператора !! (Not-null Assertion Operator)

使用 !! 操作符进行非空断言

Превращает любой тип `T?` в `T`. Если значение оказалось null, выбрасывается исключение `NullPointerException`. 将任何类型 `T?` 转换为 `T`。如果值为 null，则抛出 `NullPointerException` 异常。

```kotlin
val b: String? = null

// RU: "Я мамой клянусь, что b не null!". Если b == null -> Crash.
// CN: “我发誓 b 不是 null！”。如果 b == null -> 崩溃。
val len: Int = b!!.length
```

6. Безопасное приведение типов: оператор as?

安全类型转换：as? 操作符

Пытается привести значение к указанному типу. Если не получается — возвращает null (вместо исключения `ClassCastException`). 尝试将值转换为指定类型。如果失败，则返回 null（而不是 `ClassCastException` 异常）。

```kotlin
val obj: Any = 123

// RU: Пытаемся превратить число в строку. Не выйдет, вернется null.
// CN: 尝试将数字转换为字符串。失败，返回 null。
val str: String? = obj as? String
```

7. Хункция let

let 函数

Часто используется вместе с `?.` для выполнения блока кода только для не-null значений. 通常与 `?.` 一起使用，以便仅对非 null 值执行代码块。

```kotlin
val email: String? = "user@example.com"

// RU: Блок выполнится, только если email != null.
// CN: 仅当 email != null 时才执行代码块。
email?.let {
    // RU: Внутри блока 'it' — это не-null строка.
    // CN: 在代码块内部，'it' 是非 null 字符串。
    println("Sending to $it")
}
```

8. Свойства с отложенной инициализацией (Late-initialized Properties)

延迟初始化属性

Используется ключевое слово `lateinit var`. Это обещание компилятору, что мы инициализируем переменную позже (до первого использования). 使用 `lateinit var` 关键字。这是向编译器保证我们将稍后（在首次使用之前）初始化变量。

```kotlin
class MyService {
    // RU: Работает только с var и ссылочными типами (не Int/Double).
    // CN: 仅适用于 var 和引用类型（不适用于 Int/Double）。
    lateinit var data: String

    fun init() {
        data = "Ready"
    }
}
```

9. Расширение типов поддерживающих значение null

支持空值的类型扩展

Можно писать функции-расширения для nullable-типов (например, `String?`). Внутри функции `this` может быть null. 可以为 nullable 类型（例如 `String?`）编写扩展函数。在函数内部，`this` 可能是 null。

```kotlin
// RU: Стандартная функция Kotlin.
// CN: Kotlin 标准函数。
fun String?.isNullOrEmpty(): Boolean {
    // RU: Проверка на null происходит ВНУТРИ функции.
    // CN: Null 检查发生在函数“内部”。
    return this == null || this.length == 0
}

val s: String? = null
// RU: Можно вызывать безопасно без ?.
// CN: 可以安全调用，无需 ?.
println(s.isNullOrEmpty()) // true
```

10. Типы и подтипы, классы и подклассы

类型与子类型，类与子类

В Kotlin Класс и Тип — это не одно и то же. 在 Kotlin 中，类和类型不是一回事。

- Class: `String` (это код, который мы написали).
- Class: `String`（这是我们编写的代码）。
- Types: Из одного класса `String` получается два типа: `String` и `String?`.
- Types: 一个 `String` 类产生两种类型：`String` 和 `String?`。
- Subtype: `String` является подтипом `String?` (потому что мы можем присвоить `String` туда, где ждут `String?`).
- Subtype: `String` 是 `String?` 的子类型（因为我们可以将 `String` 赋值给期望 `String?` 的地方）。

11. Изменяемые и неизменяемые коллекции

可变与不可变集合

Kotlin строго разделяет интерфейсы коллекций. Kotlin 严格区分集合接口。

- Immutable (Read-only): `List`, `Set`, `Map`. У них нет методов `add`, `remove`.
- Immutable (只读): `List`, `Set`, `Map`. 它们没有 `add`, `remove` 方法。
- Mutable: `MutableList`, `MutableSet`, `MutableMap`. У них есть методы для изменения.
- Mutable: `MutableList`, `MutableSet`, `MutableMap`. 它们有修改方法。

```kotlin
// Read-only
val readOnly: List<Int> = listOf(1, 2, 3)

// Mutable
val mutable: MutableList<Int> = mutableListOf(1, 2, 3)
mutable.add(4)
```

9. Обобщенные типы (Generic). Параметризованные типы. Параметризованные классы. Параметризованные интерфейсы. Параметризованные функции. Хункции с параметризованным получателем. Ограничение типовых параметров - класс. Ограничение типовых параметров - интерфейс. Несколько ограничений типовых параметров. Обобщенные типы во время выполнения. Проверка и приведение для обобщенных типов. Обобщенные типы во встраиваемых функциях во время выполнения. Типы и подтипы. Вариантность и ковариантность типов (variance).

Это 9-й вопрос (Question 9), посвященный обобщениям (Generics) в Kotlin. Это сложная, но важная тема для написания гибкого и безопасного кода.
这是 第 9 题 (Question 9)，关于 Kotlin 中的泛型 (Generics)。这是一个复杂但对于编写灵活且安全的代码非常重要的主题。

1. Обобщенные типы (Generic) и Параметризованные типы

泛型与参数化类型

Обобщения позволяют определять классы и методы, которые работают с разными типами данных, не теряя строгой типизации. 泛型允许定义可以处理不同数据类型的类和方法，同时不丢失强类型检查。

```kotlin
// RU: <T> — это параметр типа. Мы можем подставить туда Int, String и т.д.
// CN: <T> 是类型参数。我们可以代入 Int, String 等。
class Box<T>(t: T) {
    var value = t
}

val box1: Box<Int> = Box(1)       // Параметризованный тип Box<Int> (参数化类型 Box<Int>)
val box2: Box<String> = Box("Hi") // Параметризованный тип Box<String> (参数化类型 Box<String>)
```

2. Параметризованные классы

参数化类

Класс, который объявляет один или несколько типовых параметров в угловых скобках `<>`. 在尖括号 `<>` 中声明一个或多个类型参数的类。

```kotlin
// RU: Класс, который может хранить пару значений любых типов.
// CN: 一个可以存储任意类型值对的类。
class Pair<A, B>(val first: A, val second: B)

val pair = Pair(1, "Apple") // A=Int, B=String
```

3. Параметризованные интерфейсы

参数化接口

Интерфейсы тоже могут быть обобщенными. Это часто используется в коллекциях (List, Set). 接口也可以是泛型的。这常用于集合（List, Set）。

```kotlin
interface Repository<T> {
    fun save(item: T)
    fun getById(id: Int): T
}

// RU: Реализация интерфейса для конкретного типа User.
// CN: 针对具体类型 User 的接口实现。
class UserRepository : Repository<User> { /*...*/ }
```

4. Параметризованные функции

参数化函数

Если класс не обобщенный, но функции нужна гибкость, мы можем сделать обобщенной только функцию. 如果类不是泛型的，但函数需要灵活性，我们可以只让函数变成泛型。

```kotlin
// RU: <T> ставится перед именем функции.
// CN: <T> 放在函数名之前。
fun <T> singletonList(item: T): List<T> {
    return listOf(item)
}
```

5. Хункции с параметризованным получателем

带参数化接收者的函数

Это функции-расширения для обобщенных типов. 这是针对泛型类型的扩展函数。

```kotlin
// RU: Расширяем любой список List<T>. 'this' имеет тип List<T>.
// CN: 扩展任何列表 List<T>。'this' 的类型是 List<T>。
fun <T> List<T>.secondOrNull(): T? {
    if (this.size < 2) return null
    return this[1]
}
```

6. Ограничение типовых параметров - класс

类型参数约束 - 类

Мы можем ограничить тип `T`, указав, что он должен быть наследником определенного класса. Используется двоеточие `:`. 我们可以通过指定 `T` 必须是特定类的子类来限制它。使用冒号 `:`。

```kotlin
// RU: T обязан быть числом (Number) или его наследником (Int, Double).
// CN: T 必须是 Number 或其子类 (Int, Double)。
fun <T : Number> doubleValue(value: T): Double {
```

```
    return value.toDouble() * 2
}
```

7. Ограничение типовых параметров - интерфейс

类型参数约束 - 接口

Синтаксис тот же. Часто используется для ограничения `Comparable`. 语法相同。常用于限制 `Comparable`。

```
// RU: Элементы должны быть сравниваемыми (Comparable).
// CN: 元素必须是可比较的 (Comparable)。
fun <T : Comparable<T>> sort(list: List<T>) { /*...*/ }
```

8. Несколько ограничений типовых параметров

多个类型参数约束

Если нужно несколько ограничений, используется ключевое слово `where` в конце. 如果需要多个约束，在末尾使用关键字 `where`。

```
// RU: T должен быть CharSequence И Comparable.
// CN: T 必须同时是 CharSequence 和 Comparable。
fun <T> copyGreaterThan(list: List<T>, threshold: T): List<String>
    where T : CharSequence,
          T : Comparable<T> {
    /*...*/
    return emptyList()
}
```

9. Обобщенные типы во время выполнения

运行时的泛型类型

В Kotlin (как и в Java) generics существуют только на этапе компиляции. В runtime происходит стирание типов (Type Erasure). 在 Kotlin 中（就像在 Java 中一样），泛型仅存在于编译阶段。在运行时会发生类型擦除 (Type Erasure)。

- `List<String>` и `List<Int>` в runtime превращаются просто в `List`.
- `List<String>` 和 `List<Int>` 在运行时都会变成 `List`。

10. Проверка и приведение для обобщенных типов

泛型类型的检查与转换

Из-за стирания типов мы не можем проверить `is List<String>`. Мы можем проверить только `is List<*>`. 由于类型擦除，我们无法检查 `is List<String>`。我们只能检查 `is List<*>`。

```
fun check(a: Any) {
    // if (a is List<String>) // Error! Cannot check for instance of erased
        type

    // RU: Star-projection (Звездная проекция). Мы проверяем, что это список
        чего-то.
    // CN: 星号投影 (Star-projection)。我们检查它是否是某种列表。
    if (a is List<*>) {
        println("It's a list")
    }
    // RU: Unchecked cast (предупреждение компилятора).
    // CN: 未经检查的转换 (编译器警告)。
    val list = a as List<String>
}
```

11. Обобщенные типы во встраиваемых функциях во время выполнения (Reified)

内联函数运行时的具体化泛型

Это уникальная фича Kotlin. Если функция `inline`, мы можем использовать `reified`, чтобы сохранить тип в runtime. 这是 Kotlin 的独特功能。如果函数是 `inline` 的，我们可以使用 `reified` 来在运行时保留类型。

```
// RU: Ключевое слово reified спасает от стирания типов.
// CN: 关键字 reified 防止了类型擦除。
inline fun <reified T> isA(value: Any): Boolean {
    // RU: Теперь мы МОЖЕМ проверить тип T!
    // CN: 现在我们“可以”检查类型 T 了！
    return value is T
}

val isString = isA<String>("Hello") // true
```

12. Типы и подтипы

类型与子类型

Важно понимать: `String` — подтип `Any`. Но `MutableList<String>` — НЕ подтип `MutableList<Any>`. 理解这一点很重要：`String` 是 `Any` 的子类型。但 `MutableList<String>` 不是 `MutableList<Any>` 的子类型。

- Если бы это было так, мы могли бы положить `Int` в список строк, что вызвало бы ошибку.
- 如果是这样的话，我们就可以把 `Int` 放进字符串列表，这会导致错误。

13. Вариантность и ковариантность типов (variance)

类型的变型与协变

Чтобы обойти ограничение из пункта 12, используются ключевые слова `out` (Covariance) и `in` (Contravariance). 为了绕过第 12 点的限制，使用关键字 `out`（协变）和 `in`（逆变）。

- **Producer (out T):** Мы только читаем T. (Covariant / 协变).
- `List<String>` является подтипом `List<Any>` (потому что `List` в Kotlin — это read-only интерфейс с `out T`).
- `List<String>` 是 `List<Any>` 的子类型（因为 Kotlin 中的 `List` 是带有 `out T` 的只读接口）。
- **Consumer (in T):** Мы только пишем T. (Contravariant / 逆变).
- `Comparable<Number>` является подтипом `Comparable<Int>`.
- `Comparable<Number>` 是 `Comparable<Int>` 的子类型。

```
// RU: out = только возвращаем (производим).
// CN: out = 只返回 (生产)。
class Producer<out T>(val value: T) {
    fun get(): T = value
}

val p: Producer<Any> = Producer<String>("test") // OK
```

10. Анонимная функция в Kotlin. Хункциональный тип. Псевдоним функционального типа. Лямбда-выражение. Лямбда-выражение для вызова функции. Лямбда-выражение как последний параметр функции. Лямбда-выражение как ссылка на функцию. Лямбда в функциях filter, map, forEach. Интерфейсы Iterable и Iterator. Хункции с предикатами all, any, count и find. Группирование элементов коллекции.

Это 10-й вопрос (Question 10), посвященный функциональному программированию в Kotlin. Это одна из самых сильных сторон языка.
这是 第 10 题 (Question 10)，关于 Kotlin 中的函数式编程。这是该语言最强大的优势之一。
Мы разберем каждую тему по порядку. 我们将按顺序解析每个主题。

1. Анонимная функция в Kotlin

Kotlin 中的匿名函数

Это функция без имени. В отличие от лямбд, здесь можно явно указать тип возвращаемого значения. 这是一个没有名字的函数。与 Lambda 不同，这里可以显式指定返回值类型。

```
// RU: Анонимная функция. Мы явно пишем 'fun' и тип возврата ': Int'.
// CN: 匿名函数。我们要显式写出 'fun' 和返回类型 ': Int'。
val multiply = fun(x: Int, y: Int): Int {
    return x * y
}
```

2. Хункциональный тип

函数类型

Переменная может хранить функцию. Тип такой переменной описывается как `(Параметры) -> Результат`. 变量可以存储函数。这种变量的类型描述为 `(参数) -> 结果`。

```
// RU: Переменная 'sum' имеет функциональный тип: принимает два Int,
       возвращает Int.
// CN: 变量 'sum' 具有函数类型：接收两个 Int，返回 Int。
val sum: (Int, Int) -> Int = { a, b -> a + b }
```

3. Псевдоним функционального типа (Type Alias)

函数类型别名

Если функциональный тип слишком длинный, ему можно дать короткое имя с помощью `typealias`. 如果函数类型太长，可以使用 `typealias` 给它起一个简短的名字。

```
// RU: Даем имя сложному типу обработчика клика.
// CN: 给复杂的点击处理程序类型起个名字。
typealias ClickHandler = (Button, Event) -> Unit

fun setOnClick(handler: ClickHandler) { /*...*/ }
```

4. Лямбда-выражение

Lambda 表达式

Это краткий способ записи анонимной функции. Всегда пишется в фигурных скобках `{}`. 这是编写匿名函数的简洁方式。总是写在花括号 `{}` 中。

```
// RU: Лямбда. Параметры слева от ->, тело справа.
// CN: Lambda。参数在 -> 左边，函数体在右边。
val greet = { name: String -> println("Hello, $name") }
```

5. Лямбда-выражение для вызова функции

用于调用函数的 Lambda 表达式

Лямбду можно вызвать методом `invoke()` или просто как обычную функцию `()`. 可以通过 `invoke()` 方法或直接像普通函数 `()` 一样调用 Lambda。

```
val square = { x: Int -> x * x }

// Способ 1 (Method 1)
println(square(5))

// Способ 2 (Method 2)
println(square.invoke(5))
```

6. Лямбда-выражение как последний параметр функции (Trailing Lambda)

作为函数最后一个参数的 Lambda 表达式（尾随 Lambda）

Если лямбда — это последний аргумент функции, её можно вынести за скобки. Это “фирменный стиль” Kotlin. 如果 Lambda 是函数的最后一个参数，可以将其移至括号外。这是 Kotlin 的“标志性风格”。

```
fun runAsync(delay: Int, action: () -> Unit) { /*...*/ }

// RU: Лямбда вынесена за круглые скобки. Выглядит как встроенная
       конструкция языка.
// CN: Lambda 被移到了圆括号外面。看起来像是语言的内置结构。
runAsync(1000) {
    println("Done!")
}
```

7. Лямбда-выражение как ссылка на функцию

作为函数引用的 Lambda 表达式

Вместо того чтобы писать `{ x -> func(x) }`, можно использовать ссылку на уже существующую функцию через `::`. 与其写 `{ x -> func(x) }`，不如通过 `::` 使用对现有函数的引用。

```
fun isEven(x: Int) = x % 2 == 0

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

```
// RU: Передаем ссылку на функцию isEven.
// CN: 传递对函数 isEven 的引用。
list.filter(::isEven)
```

8. Лямбда в функциях filter, map, forEach

filter, map, forEach 函数中的 Lambda

Это стандартные функции высшего порядка для работы с коллекциями. 这些是用于处理集合的标准高阶函数。

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

// 1. filter: Оставляет только то, что true.
// 1. filter: 只保留 true 的项。
// RU: 'it' — это неявное имя единственного параметра лямбды.
// CN: 'it' 是 Lambda 唯一参数的隐式名称。
val evens = list.filter { it % 2 == 0 }

// 2. map: Преобразует каждый элемент.
// 2. map: 转换每个元素。
val squared = list.map { it * it }

// 3. forEach: Действие для каждого элемента.
// 3. forEach: 对每个元素执行操作。
list.forEach { println(it) }
```

9. Интерфейсы Iterable и Iterator

Iterable 和 Iterator 接口

`Iterable` — это коллекция, по которой можно пройтись. `Iterator` — это объект-курсор, который делает шаги. `Iterable` 是可以遍历的集合。`Iterator` 是执行步进操作的光标对象。

- **Iterable:** Имеет метод `iterator()`. (拥有 `iterator()` 方法).
- **Iterator:** Имеет методы `hasNext()` и `next()`. (拥有 `hasNext()` 和 `next()` 方法).

10. Хункции с предикатами all, any, count и find

带有谓词 all, any, count 和 find 的函数

Предикат — это лямбда, возвращающая `Boolean`. 谓词是返回 `Boolean` 的 Lambda。

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

// RU: Все элементы больше 0? -> true
// CN: 所有元素都大于 0 吗？-> true
val allPositive = numbers.all { it > 0 }

// RU: Есть ли хоть один элемент больше 10? -> false
// CN: 有没有至少一个元素大于 10？-> false
val hasBigNumber = numbers.any { it > 10 }

// RU: Сколько четных чисел? -> 2
// CN: 有多少个偶数？-> 2
val evenCount = numbers.count { it % 2 == 0 }

// RU: Найти первый элемент больше 3. -> 4
// CN: 找到第一个大于 3 的元素。-> 4
val firstBig = numbers.find { it > 3 }
```

11. Группирование элементов коллекции

集合元素分组

Хункция `groupBy` превращает список в карту (`Map`), где ключ — это критерий группировки. `groupBy` 函数将列表转换为映射（`Map`），其中键是分组标准。

```
val words = listOf("a", "abc", "ab", "def", "abcd")

// RU: Группируем слова по их длине.
// CN: 按长度对单词进行分组。
val byLength: Map<Int, List<String>> = words.groupBy { it.length }

// Result:
// 1 -> ["a"]
// 2 -> ["ab"]
// 3 -> ["abc", "def"]
// 4 -> ["abcd"]
```

11. Хункция, анонимная функция, лямбда. Хункция-расширение, анонимная функция-расширение, лямбда с получателем. Хункциональный тип с получателем. Лямбда-выражение с получателем. Стандартная функция with. Стандартная функция apply. Стандартная функция let. Стандартная функция run. Объявление функциональных интерфейсов в Kotlin 1.4.

Это 11-й вопрос (Question 11), который углубляется в тему функций и контекста (Receiver) в Kotlin. Это база для понимания DSL (Domain Specific Languages) и стандартной библиотеки.
这是 第 11 题 (Question 11)，深入探讨 Kotlin 中的函数和上下文 (Receiver)。这是理解 DSL（领域特定语言）和标准库的基础。
Мы разберем каждое понятие по порядку. 我们将按顺序解析每个概念。

1. Хункция, анонимная функция, лямбда

函数，匿名函数，Lambda

Это три способа определить блок кода. 这是定义代码块的三种方式。

- **Хункция (Function):** Именованная, объявлена через `fun`.
- **函数 (Function):** 具名的，通过 `fun` 声明。
- **Анонимная функция (Anonymous Function):** `fun(...)` без имени. Позволяет явно указать `return`.
- **匿名函数 (Anonymous Function):** 没有名字的 `fun(...)`。允许显式指定 `return`。
- **Лямбда (Lambda):** Выражение в `{}`. Самый краткий синтаксис.
- **Lambda:** `{}` 中的表达式。语法最简洁。

```
// 1. Обычная функция (Regular function)
fun add(x: Int, y: Int): Int = x + y

// 2. Анонимная функция (Anonymous function)
// RU: Используется, когда нужен явный return или сложная логика возврата.
// CN: 当需要显式 return 或复杂的返回逻辑时使用。
val anon = fun(x: Int, y: Int): Int {
    return x + y
}

// 3. Лямбда (Lambda)
// RU: Самый частый вариант.
// CN: 最常见的选项。
val lambda = { x: Int, y: Int -> x + y }
```

2. Хункция-расширение, анонимная функция-расширение, лямбда с получателем

扩展函数，匿名扩展函数，带接收者的 Lambda

Эти механизмы позволяют вызывать функцию так, будто она является частью класса. 这些机制允许像调用类的一部分一样调用函数。

- Хункция-расширение: `fun String.lastChar()`
- 扩展函数: `fun String.lastChar()`
- Анонимная функция-расширение: `fun String.() { ... }`
- 匿名扩展函数: `fun String.() { ... }`
- Лямбда с получателем: Блок кода, где `this` указывает на расширяемый объект.
- 带接收者的 Lambda: 代码块，其中 `this` 指向被扩展的对象。

3. Хункциональный тип с получателем

带接收者的函数类型

Это тип переменной, который описывает, что функция должна вызываться на определенном объекте. 这是变量的类型，描述了函数必须在特定对象上调用。
Синтаксис (Syntax): `Receiver.(Params) -> ReturnType`

```
// RU: Тип переменной: функция, которая вызывается на Int и ничего не
        возвращает.
// CN: 变量类型：在 Int 上调用且不返回任何内容的函数。
val sum: Int.(Int) -> Int
```

4. Лямбда-выражение с получателем

带接收者的 Lambda 表达式

Это сама реализация функционального типа с получателем. Внутри лямбды объект доступен через `this`. 这是带接收者的函数类型的实现本身。在 Lambda 内部，可以通过 `this` 访问对象。

```
// RU: Реализация типа из пункта 3.
// CN: 第 3 点中类型的实现。
val sum: Int.(Int) -> Int = { other ->
    // RU: 'this' — это первое число (получатель), 'other' — аргумент.
    // CN: 'this' 是第一个数字（接收者），'other' 是参数。
    this + other
}

val result = 1.sum(2) // 3
```

5. Стандартная функция with

标准函数 with

Принимает объект и лямбду с получателем. Используется для группировки вызовов методов одного объекта. Не является расширением. 接收一个对象和一个带接收者的 Lambda。用于对同一个对象的方法调用进行分组。不是扩展函数。

- **Context:** `this`
- **Return:** Результат лямбды (Result of lambda).

```
val sb = StringBuilder()

// RU: "С этим объектом (sb) сделай следующее..."
// CN: “对这个对象 (sb) 做以下事情……”
val result = with(sb) {
    append("Hello")
    append(" World")
    this.toString() // Return value
}
```

6. Стандартная функция apply

标准函数 apply

Используется для настройки (инициализации) объекта. Возвращает сам объект. 用于配置（初始化）对象。返回对象本身。

- **Context:** `this`
- **Return:** Объект (The object itself).

```
// RU: Создаем объект и сразу настраиваем его поля.
// CN: 创建对象并立即配置其字段。
val person = Person().apply {
    name = "John"  // this.name = "John"
    age = 30
} // person variable = initialized object
```

7. Стандартная функция let

标准函数 let

Используется для выполнения действий с результатом выражения (часто для проверки на null). 用于对表达式的结果执行操作（常用于 null 检查）。

- **Context:** `it` (аргумент / argument)
- **Return:** Результат лямбды (Result of lambda).

```
val str: String? = "Hello"

// RU: Выполнится, только если str не null.
// CN: 仅当 str 不为 null 时执行。
str?.let {
    // RU: Здесь доступ через 'it'.
    // CN: 这里通过 'it' 访问。
    println(it.length)
}
```

8. Стандартная функция run
标准函数 run
Это комбинация `with` и `let`. 这是 `with` 和 `let` 的结合体。
1. Как расширение (object.run): Похож на `let`, но внутри `this`. (Контекст `this`, возвращает результат лямбды).
2. Без расширения (run { ⋯ }): Просто запускает блок кода.

```
val service = Service()

// RU: Настраиваем и вычисляем результат (в отличие от apply, который вернул бы service).
// CN: 配置并计算结果（与 apply 不同，apply 会返回 service）。
val initResult = service.run {
    port = 8080
    start() // returns Boolean
}
```

9. Объявление функциональных интерфейсов в Kotlin 1.4
Kotlin 1.4 中函数式接口的声明
Раньше SAM-конверсия (Single Abstract Method) работала только для Java-интерфейсов. Начиная с версии 1.4, можно объявлять SAM-интерфейсы в Kotlin, используя ключевое слово `fun interface`. 以前，SAM 转换（单抽象方法）仅适用于 Java 接口。从 1.4 版本开始，可以使用关键字 `fun interface` 在 Kotlin 中声明 SAM 接口。

```
// RU: Ключевое слово fun перед interface.
// CN: interface 前面的 fun 关键字。
fun interface Predicate {
    fun accept(i: Int): Boolean
}

// RU: Теперь можно использовать лямбду вместо создания объекта.
// CN: 现在可以使用 Lambda 代替创建对象。
val isEven = Predicate { it % 2 == 0 }
```

12. Виды коллекций. Изменяемые и не изменяемые. Итераторы и итерируемые. Изменяемая коллекция. Виды коллекций. List (Список). Изменяемый список. Виды коллекций. Set (Множество). Виды коллекций. Map (Карта). Конструирование множеств. Конструирование списков. Конструирование карт. Итераторы над коллекциями.

Это 12-й вопрос (Question 12), посвященный коллекциям в Kotlin. 这是 第 12 题 (Question 12)，关于 Kotlin 中的集合。
Мы разберем каждое понятие по порядку. 我们将按顺序解析每个概念。

1. Виды коллекций
集合的类型
В стандартной библиотеке Kotlin основные типы коллекций — это List (Список), Set (Множество) и Map (Карта/Словарь). 在 Kotlin 标准库中，主要的集合类型是 List（列表）、Set（集）和 Map（映射/字典）。

2. Изменяемые и не изменяемые
可变与不可变
Kotlin строго разделяет коллекции на два типа интерфейсов:
1. **Read-only (Immutable):** Только для чтения (нет методов `add`, `remove`).
2. **Mutable:** Изменяемые (наследуются от Read-only и добавляют методы изменения).

Kotlin 严格将集合分为两种类型的接口：
1. **Read-only (不可变):** 只读（没有 `add`、`remove` 方法）。
2. **Mutable (可变):** 可变（继承自 Read-only 并添加了修改方法）。

3. Итераторы и итерируемые
迭代器与可迭代对象
- **Iterable:** Интерфейс, который имеет метод `iterator()`. По нему можно запустить цикл `for`.
- **Iterator:** Объект, который выполняет обход, имея методы `hasNext()` и `next()`.
- **Iterable:** 拥有 `iterator()` 方法的接口。可以在其上运行 `for` 循环。
- **Iterator:** 执行遍历的对象，拥有 `hasNext()` 和 `next()` 方法。

4. Изменяемая коллекция (MutableCollection)
可变集合
Это общий интерфейс для коллекций, которые можно менять. Он добавляет методы `add`, `remove`, `clear`. 这是可修改集合的通用接口。它添加了 `add`、`remove`、`clear` 方法。

```
fun modify(collection: MutableCollection<String>) {
    // RU: Мы можем добавлять, так как это Mutable.
    // CN: 我们可以添加，因为这是 Mutable。
    collection.add("New Item")
}
```

5. List (Список)
列表
Упорядоченная коллекция. Элементы имеют индексы (0, 1, 2⋯). Дубликаты разрешены.
Интерфейс `List` — только для чтения. 有序集合。元素具有索引 (0, 1, 2⋯)。允许重复。`List` 接口仅供读取。

```
// RU: Нельзя добавить элементы сюда.
// CN: 不能往这里添加元素。
val list: List<Int> = listOf(1, 2, 3)
val item = list[0] // Access by index
```

6. Изменяемый список (MutableList)
可变列表
Наследник `List`, который позволяет изменять элементы по индексу, добавлять и удалять их. `List` 的子接口，允许通过索引修改元素，以及添加和删除元素。

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
// RU: Замена элемента по индексу.
// CN: 通过索引替换元素。
numbers[0] = 10
numbers.add(4)
```

7. Set (Множество)
集 (Set)
Коллекция уникальных элементов. Порядок элементов не гарантирован (в общем случае). 唯一元素的集合。通常不保证元素的顺序。

```
// RU: Единица добавится только один раз.
// CN: 数字 1 只会被添加一次。
val set = setOf(1, 2, 1, 3) // {1, 2, 3}
```

8. Map (Карта)
映射 (Map)
Набор пар "Ключ-Значение" (Key-Value). Ключи уникальны, значения могут повторяться. Важно: `Map` не наследуется от интерфейса `Collection`. "键-值"对 (Key-Value) 的集合。键是唯一的，值可以重复。 重要提示： `Map` 不继承自 `Collection` 接口。

9. Конструирование множеств
构建 Set
Используются функции `setOf()` (для неизменяемых) и `mutableSetOf()` (для изменяемых). 使用 `setOf()`（用于不可变）和 `mutableSetOf()`（用于可变）函数。

```
val readOnlySet = setOf("A", "B")
val mutableSet = mutableSetOf("A", "B")
mutableSet.add("C")
```

10. Конструирование списков
构建 List
Используются функции `listOf()` и `mutableListOf()`. Самая частая операция. 使用 `listOf()` 和 `mutableListOf()` 函数。这是最常见的操作。

```
val empty = emptyList<String>()
val values = listOf(1, 2, 3)
// RU: ArrayList под капотом.
// CN: 底层是 ArrayList。
val arrayList = arrayListOf(1, 2, 3)
```

11. Конструирование карт
构建 Map
Используются функции `mapOf()` и `mutableMapOf()`. Для создания пар используется инфиксная функция `to`. 使用 `mapOf()` 和 `mutableMapOf()` 函数。使用中缀函数 `to` 创建键值对。

```
// RU: "Key" to "Value".
// CN: "Key" to "Value".
val map = mapOf(
    1 to "One",
    2 to "Two"
)
```

12. Итераторы над коллекциями
集合上的迭代器
Вы можете получить итератор вручную, но чаще используется цикл `for`. 您可以手动获取迭代器，但更常用的是 `for` 循环。

```
val list = listOf("A", "B")

// Способ 1: Явный итератор (Explicit Iterator)
val iterator = list.iterator()
while (iterator.hasNext()) {
    println(iterator.next())
}

// Способ 2: Цикл for (скрывает итератор)
// 方法 2: for 循环 (隐藏了迭代器)
for (item in list) {
    println(item)
}
```

13. Преобразования коллекций. Mapping. Преобразования карт. Mapping. Преобразования коллекций. Zipping. Преобразования коллекций. Association. Преобразования коллекций. Flattening. Хильтрация коллекций. predicate. Хильтрация коллекций. Partitioning. Хильтрация коллекций. Testing predicates. Операторы + и - для работы с коллекциями. Группирование. groupBy. Восстановление частей коллекции. Slice. Восстановление частей коллекции. take & drop.

Это 13-й вопрос (Question 13), посвященный мощным инструментам преобразования коллекций в Kotlin. Это основа функционального стиля работы с данными.
这是 第 13 题 (Question 13)，关于 Kotlin 中强大的集合转换工具。这是处理数据的函数式风格的基础。
Мы разберем каждый пункт по порядку. 我们将按顺序解析每一点。

1. Преобразования коллекций. Mapping.
集合转换：Mapping (映射)
Хункция `map` создает новую коллекцию, применяя функцию-трансформер к каждому элементу. `map` 函数通过对每个元素应用转换函数来创建一个新集合。

```
val numbers = listOf(1, 2, 3)
```

```
// RU: Возводим каждое число в квадрат.
// CN: 将每个数字平方。
val squared = numbers.map { it * it } // [1, 4, 9]
```

2. Преобразования карт. Mapping.

Map (映射) 的转换

Для `Map` есть специальные функции: `mapKeys` (изменяет ключи) и `mapValues` (изменяет значения). 对于 `Map`，有特殊的函数：`mapKeys`（更改键）和 `mapValues`（更改值）。

```
val numbersMap = mapOf("key1" to 1, "key2" to 2)

// RU: Изменяем только значения (умножаем на 10).
// CN: 只更改值（乘以 10）。
val newMap = numbersMap.mapValues { entry -> entry.value * 10 }
// Result: {key1=10, key2=20}
```

3. Преобразования коллекций. Zipping.

集合转换：Zipping (配对)

Хункция `zip` объединяет две коллекции в одну, создавая список пар (`Pair`). Длина результата равна длине самой короткой коллекции. `zip` 函数将两个集合合并为一个，创建对 (`Pair`) 的列表。结果的长度等于最短集合的长度。

```
val names = listOf("Alice", "Bob")
val ages = listOf(20, 30, 40)

// RU: "Alice" с 20, "Bob" с 30. 40 игнорируется.
// CN: "Alice" 配 20，"Bob" 配 30。40 被忽略。
val pairs = names.zip(ages)
// Result: [("Alice", 20), ("Bob", 30)]
```

4. Преобразования коллекций. Association.

集合转换：Association (关联)

Позволяет создать `Map` из списка.

- `associateWith`: Элементы списка становятся ключами.
- `associateBy`: Элементы списка становятся значениями.

允许从列表创建 `Map`。

- `associateWith`: 列表元素成为键。
- `associateBy`: 列表元素成为值。

```
val keys = listOf("a", "b")

// RU: Ключи - буквы, значения - их длина.
// CN: 键是字母，值是它们的长度。
val map = keys.associateWith { it.length }
// { "a"=1, "b"=1 }
```

5. Преобразования коллекций. Flattening.

集合转换：Flattening (扁平化)

Превращает вложенные коллекции (список списков) в один плоский список.

- `flatten()`: Просто объединяет.
- `flatMap()`: Сначала преобразует (map), потом объединяет (flatten).

将嵌套集合（列表的列表）转换为一个扁平列表。

- `flatten()`: 只是合并。
- `flatMap()`: 先转换 (map)，后合并 (flatten)。

```
val nested = listOf(listOf(1, 2), listOf(3, 4))

// RU: Результат: [1, 2, 3, 4]
// CN: 结果：[1, 2, 3, 4]
val flat = nested.flatten()
```

6. Хильтрация коллекций. predicate.

集合过滤：predicate (谓词)

Предикат — это условие (лямбда), возвращающее `true` или `false`. Хункция `filter` оставляет только те элементы, для которых предикат вернул `true`. 谓词是返回 `true` 或 `false` 的条件 (Lambda)。`filter` 函数只保留谓词返回 `true` 的元素。

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

// RU: Оставляем только четные.
// CN: 只保留偶数。
val evens = list.filter { it % 2 == 0 } // [2, 4]
```

7. Хильтрация коллекций. Partitioning.

集合过滤：Partitioning (分区)

Разделяет коллекцию на две: одна с элементами, подходящими под условие, вторая — с остальными. 将集合分为两个：一个包含符合条件的元素，另一个包含其余元素。

```
// RU: Результат - это Пара (Pair) из двух списков.
// CN: 结果是由两个列表组成的对 (Pair)。
val (match, rest) = list.partition { it > 2 }

// match: [3, 4]
// rest: [1, 2]
```

8. Хильтрация коллекций. Testing predicates.

集合过滤：Testing predicates (测试谓词)

Проверка элементов на соответствие условию (возвращает Boolean).

- `any()`: Фотя бы один.
- `all()`: Все.
- `none()`: Ни один.

检查元素是否符合条件（返回 Boolean）。

- `any()`: 至少一个。
- `all()`: 所有。
- `none()`: 没有一个。

```
val list = listOf(1, 2, 3)
val hasEven = list.any { it % 2 == 0 } // true
```

9. Операторы + и - для работы с коллекциями

用于集合的 + 和 - 操作符

Эти операторы создают новую коллекцию, добавляя или удаляя элементы (не меняя исходную). 这些操作符创建新集合，添加或删除元素（不更改原始集合）。

```
val list = listOf("A")

// RU: Создается новый список.
// CN: 创建一个新列表。
val plusList = list + "B" // ["A", "B"]
val minusList = plusList - "A" // ["B"]
```

10. Группирование. groupBy.

分组：groupBy

Группирует элементы в `Map`, где ключ — это критерий группировки, а значение — список элементов. 将元素分组到 `Map` 中，其中键是分组标准，值是元素列表。

```
val words = listOf("one", "two", "three")

// RU: Группируем по первой букве.
// CN: 按首字母分组。
val byFirstChar = words.groupBy { it.first() }
// Result: { 'o'=[one], 't'=[two, three] }
```

11. Восстановление частей коллекции. Slice.

获取集合部分：Slice (切片)

Возвращает список элементов по указанным индексам (диапазону или набору). 返回指定索引（范围或集合）处的元素列表。

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

// RU: Элементы с индексами от 1 до 3.
// CN: 索引从 1 到 3 的元素。
val sub = list.slice(1..3) // ["b", "c", "d"]
```

12. Восстановление частей коллекции. take & drop.

获取集合部分：take & drop (取与丢)

- `take(n)`: Взять первые `n` элементов.
- `drop(n)`: Пропустить первые `n` элементов и вернуть остальные.
- `takeLast(n)` / `dropLast(n)`: То же самое, но с конца.
- `take(n)`: 取前 `n` 个元素。
- `drop(n)`: 跳过前 `n` 个元素并返回其余元素。
- `takeLast(n)` / `dropLast(n)`: 同样的操作，但是从末尾开始。

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

val firstTwo = list.take(2) // [1, 2]
val withoutFirst = list.drop(1) // [2, 3, 4]
```

14. Получение элемента. По позиции. Получение случайного элемента. Проверка существования элемента. Упорядочение коллекции. Собственный порядок коллекции. Обратный порядок коллекции. Случайный порядок коллекции. Агрегатные операции. min, sum, average. Агрегатные операции. maxBy, maxWith. Хункции fold и reduce. Хункции foldIndexed и foldRightIndexed. Изменения в коллекции. Добавление. Изменения в коллекции. Удаление.

Это 14-й вопрос (Question 14), посвященный операциям получения данных, сортировки, агрегации и модификации коллекций в Kotlin.
这是 第 14 题 (Question 14)，关于 Kotlin 中集合的数据获取、排序、聚合和修改操作。
Мы разберем каждый пункт по порядку. 我们将按顺序解析每一点。

1. Получение элемента. По позиции.

获取元素：按位置

Для списков (`List`) элементы можно получать по индексу. 对于列表 (`List`)，可以通过索引获取元素。

```
val list = listOf("A", "B", "C")

// 1. Оператор [] (The [] operator)
val item1 = list[0]

// 2. Безопасное получение (Safe retrieval)
// RU: Если индекса нет, вернет null (не упадет).
// CN: 如果索引不存在，返回 null（不会崩溃）。
val item2 = list.getOrNull(10)

// 3. Значение по умолчанию (Default value)
// RU: Если индекса нет, вернет "Unknown".
// CN: 如果索引不存在，返回 "Unknown"。
val item3 = list.getOrElse(10) { "Unknown" }
```

2. Получение случайного элемента.

获取随机元素

Используются функции `random()` или `randomOrNull()`. 使用 `random()` 或 `randomOrNull()` 函数。

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

// RU: Вернет случайное число из списка.
// CN: 返回列表中的一个随机数。
val randomItem = list.random()
```

3. Проверка существования элемента.

检查元素是否存在

Используется оператор `in` или метод `contains()`. Также `isEmpty()` для проверки пустоты.
使用 `in` 操作符或 `contains()` 方法。还有用于检查是否为空的 `isEmpty()`。

```
val list = listOf("A", "B")

// RU: Оператор in (более идиоматично).
// CN: in 操作符（更符合惯用法）。
val exists = "A" in list // true

// RU: Метод contains.
// CN: contains 方法。
val exists2 = list.contains("Z") // false
```

4. Упорядочение коллекции.

集合排序

Стандартная сортировка (по возрастанию) для типов, реализующих `Comparable` (числа, строки).
针对实现了 `Comparable` 的类型（数字、字符串）的标准排序（升序）。

```
val numbers = listOf(3, 1, 2)

// RU: Возвращает НОВЫЙ отсортированный список.
// CN: 返回一个排好序的“新”列表。
val sorted = numbers.sorted() // [1, 2, 3]
```

5. Собственный порядок коллекции.

自定义集合顺序

Используется функция `sortedBy` (сортировка по какому-то свойству) или `sortedWith` (Comparator). 使用 `sortedBy` 函数（按属性排序）或 `sortedWith` (Comparator)。

```
val words = listOf("Banana", "Apple", "Pear")

// RU: Сортировка по длине слова.
// CN: 按单词长度排序。
val byLength = words.sortedBy { it.length }
// [Pear, Apple, Banana] (4, 5, 6 letters)
```

6. Обратный порядок коллекции.

集合反序

Можно просто перевернуть список (`reversed`) или отсортировать по убыванию (`sortedDescending`). 可以直接反转列表 (`reversed`) 或按降序排序 (`sortedDescending`)。

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

// RU: Просто переворачивает список задом наперед.
// CN: 只是将列表倒过来。
val reversed = list.reversed() // [3, 2, 1]

// RU: Сортирует от большего к меньшему.
// CN: 从大到小排序。
val descending = list.sortedDescending()
```

7. Случайный порядок коллекции.

集合随机顺序

Используется функция `shuffled()`. Это “перетасовка” колоды. 使用 `shuffled()` 函数。这是像“洗牌”一样的操作。

```
val list = listOf(1, 2, 3)
// RU: Каждый раз разный порядок.
// CN: 每次顺序都不同。
val mixed = list.shuffled()
```

8. Агрегатные операции. min, sum, average.

聚合操作：min, sum, average

Операции, которые сводят коллекцию к одному значению.

- `minOrNull()`: Минимальный элемент (в старых версиях был `min`).
- `sum()`: Сумма (только для чисел).
- `average()`: Среднее значение (Double).

将集合规约为单个值的操作。

- `minOrNull()`: 最小元素（旧版本中是 `min`）。
- `sum()`: 求和（仅限数字）。
- `average()`: 平均值 (Double)。

9. Агрегатные операции. maxBy, maxWith.

聚合操作：maxBy, maxWith

Поиск максимального элемента по критерию. 按标准查找最大元素。

- `maxByOrNull { selector }`: Максимум по возвращаемому значению селектора.
- `maxWithOrNull (Comparator)`: Максимум с использованием компаратора.
- `maxByOrNull { selector }`: 根据选择器返回值求最大值。
- `maxWithOrNull (Comparator)`: 使用比较器求最大值。

```
val list = listOf("a", "abc", "ab")

// RU: Самая длинная строка.
// CN: 最长的字符串。
val longest = list.maxByOrNull { it.length } // "abc"
```

10. Хункции fold и reduce.

fold 和 reduce 函数

Эти функции последовательно применяют операцию к элементам и накапливают результат. 这些函数依次对元素应用操作并累积结果。

- **reduce:** Первый элемент становится начальным значением аккумулятора. (Пустой список вызовет ошибку).
- **fold:** Принимает начальное значение аккумулятора явно. (Безопасен для пустого списка).
- **reduce:** 第一个元素成为累加器的初始值。（空列表会导致错误）。
- **fold:** 显式接收累加器的初始值。（对空列表安全）。

```
val numbers = listOf(1, 2, 3)

// RU: Начинаем с 0. 0+1 -> 1+2 -> 3+3 = 6.
// CN: 从 0 开始。0+1 -> 1+2 -> 3+3 = 6。
val sumFold = numbers.fold(0) { acc, item -> acc + item }
```

11. Хункции foldIndexed и foldRightIndexed.

foldIndexed 和 foldRightIndexed 函数

- `foldIndexed`: То же, что `fold`, но лямбда получает индекс элемента.
- `foldRightIndexed`: Итерация идет с конца списка к началу (справа налево).
- `foldIndexed`: 与 `fold` 相同，但 Lambda 会接收元素的索引。
- `foldRightIndexed`: 从列表末尾向前迭代（从右到左）。

```
val list = listOf("A", "B")

// RU: acc — аккумулятор, i — индекс, s — строка.
// CN: acc — 累加器, i — 索引, s — 字符串。
val res = list.foldIndexed("") { i, acc, s ->
    "$acc $i:$s"
}
// Result: " 0:A 1:B"
```

12. Изменения в коллекции. Добавление.

集合修改：添加

Работает только с `MutableList`. 仅适用于 `MutableList`。

```
val mutableList = mutableListOf(1, 2)

// RU: Добавить один элемент.
// CN: 添加一个元素。
mutableList.add(3)

// RU: Добавить другую коллекцию (addAll).
// CN: 添加另一个集合 (addAll)。
mutableList += listOf(4, 5) // [1, 2, 3, 4, 5]
```

13. Изменения в коллекции. Удаление.

集合修改：删除

Работает только с `MutableList`. 仅适用于 `MutableList`。

```
val list = mutableListOf("A", "B", "C")

// RU: Удалить конкретный объект (вернет true, если удалил).
// CN: 删除特定对象（如果删除了则返回 true）。
list.remove("B")

// RU: Удалить по индексу.
// CN: 按索引删除。
list.removeAt(0) // Удалит "A"
```

15. Списки. Доступ по индексу. Линейный поиск. Двоичный поиск в сортированных списках. Двоичный поиск с компаратором. Двоичный поиск с функцией сравнения. Добавление и обновление. Удаление. Сортировка.

Это 15-й вопрос (Question 15). Я повторю его разбор, но на этот раз без картинок, чтобы вам было удобно копировать текст.
这是 第 15 题 (Question 15)。我会重新解析一遍，但这次不包含图片，以便您复制文本。

1. Списки

列表

В Kotlin списки (`List`) — это упорядоченные коллекции элементов. Каждый элемент имеет свою позицию (индекс). 在 Kotlin 中，列表 (`List`) 是元素的有序集合。每个元素都有自己的位置（索引）。

2. Доступ по индексу

按索引访问

Индексация начинается с нуля. Для доступа используются квадратные скобки `[]` или метод `get()`. 索引从零开始。使用方括号 `[]` 或 `get()` 方法进行访问。

```
val list = listOf("A", "B", "C")

// RU: Получаем элемент с индексом 1.
// CN: 获取索引为 1 的元素。
val item = list[1] // "B"
```

3. Линейный поиск

线性搜索

Это стандартный поиск, который проверяет элементы один за другим. Работает на любых списках, но медленно (O(n)). 这是标准的搜索方式，逐个检查元素。适用于任何列表，但速度较慢 (O(n))。

```
val numbers = listOf(10, 20, 30, 20)

// RU: Найти индекс первого вхождения числа 20.
// CN: 查找数字 20 第一次出现的索引。
val index = numbers.indexOf(20) // 1

// RU: Найти индекс последнего вхождения.
// CN: 查找最后一次出现的索引。
val lastIndex = numbers.lastIndexOf(20) // 3
```

4. Двоичный поиск в сортированных списках

排序列表中的二分查找

Работает только в отсортированных списках. Делит список пополам, поэтому работает очень быстро (O(log n)). 仅适用于已排序的列表。它将列表一分为二，因此速度非常快 (O(log n))。

```
val sortedList = listOf(1, 3, 5, 7, 9)

// RU: Ищем число 7.
```

```
// CN: 搜索数字 7。
val index = sortedList.binarySearch(7) // 3
```

---

5. Двоичный поиск с компаратором
带比较器的二分查找
Используется, если нужно искать объекты (не числа/строки) или если сортировка нестандартная.
当需要搜索对象（非数字/字符串）或排序非标准时使用。

```
val people = listOf(Person("Alice", 20), Person("Bob",
        30)) // Сортирован по возрасту (按年龄排序)

// RU: Ищем человека с возрастом 30.
// CN: 搜索年龄为 30 的人。
val index = people.binarySearch(Person("Any", 30), Comparator { p1, p2 ->
    p1.age - p2.age
})
```

---

6. Двоичный поиск с функцией сравнения
带比较函数的二分查找
Самый гибкий способ. Вы передаете функцию, которая возвращает `Int`:
- `0`: Нашли.
- `<0`: Искомое значение меньше текущего (искать слева).
- `>0`: Искомое значение больше текущего (искать справа).

最灵活的方式。传递一个返回 `Int` 的函数：
- `0`: 找到了。
- `<0`: 目标值小于当前值（向左搜）。
- `>0`: 目标值大于当前值（向右搜）。

```
val list = listOf(2, 4, 6, 8)

// RU: Ищем число 6.
// CN: 搜索数字 6。
val index = list.binarySearch { item ->
    item.compareTo(6)
}
```

---

7. Добавление и обновление
添加和更新
Возможно только в изменяемых списках (`MutableList`). 仅在可变列表 (`MutableList`) 中可用。

```
val list = mutableListOf("One", "Two")

// RU: Добавление в конец.
// CN: 添加到末尾。
list.add("Three")

// RU: Вставка по индексу (сдвигает остальные элементы).
// CN: 按索引插入（移动其他元素）。
list.add(0, "Zero")

// RU: Обновление (замена) значения.
// CN: 更新（替换）值。
list[1] = "Updated One"
```

---

8. Удаление
删除
Возможно только в изменяемых списках (`MutableList`). 仅在可变列表 (`MutableList`) 中可用。

```
val list = mutableListOf("A", "B", "C")

// RU: Удаляет первый найденный элемент "B".
// CN: 删除找到的第一个元素 "B"。
list.remove("B")

// RU: Удаляет элемент по индексу 0.
// CN: 删除索引为 0 的元素。
list.removeAt(0)
```

---

9. Сортировка
排序
- `sort()`: Сортирует текущий список (только MutableList).
- `sorted()`: Создает новый отсортированный список (для любых List).
- `sort()`: 对当前列表进行排序（仅限 MutableList）。
- `sorted()`: 创建一个排好序的新列表（适用于任何 List）。

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

// RU: Изменяет сам список numbers.
// CN: 修改 numbers 列表本身。
numbers.sort() // [1, 2, 3]

// RU: Сортировка по убыванию.
// CN: 降序排序。
numbers.sortDescending() // [3, 2, 1]
```

---

16. Множества. Хункции union, intersect и subtract. Карты. Получение ключей и значений. Хильтрация. Операции + и -. Добавление и обновление. Удаление.

Это 16-й вопрос (Question 16), посвященный операциям над множествами (Set) и картами (Map) в Kotlin. 这是 第 16 题 (Question 16)，关于 Kotlin 中集合 (Set) 和映射 (Map) 的操作。
Мы разберем каждый пункт подробно без использования картинок. 我们将详细解析每一点，不使用图片。

---

1. Множества (Sets)
集合
Коллекция, содержащая только уникальные элементы. Порядок элементов, как правило, не гарантируется (кроме `LinkedHashSet`). 仅包含唯一元素的集合。通常不保证元素的顺序（`LinkedHashSet` 除外）。

---

2. Хункции union, intersect и subtract
union, intersect 和 subtract 函数
Это операции теории множеств, возвращающие новое множество. 这是集合论操作，返回一个新集合。
- **union (Объединение / 并集):** Элементы из обоих множеств (уникальные).
- **intersect (Пересечение / 交集):** Только те элементы, которые есть в обоих множествах.
- **subtract (Разность / 差集):** Элементы первого множества, которых нет во втором.

```
val set1 = setOf(1, 2, 3)
val set2 = setOf(3, 4, 5)

// RU: Объединение (1, 2, 3, 4, 5).
// CN: 并集 (1, 2, 3, 4, 5)。
val u = set1.union(set2) // или set1 + set2

// RU: Пересечение (3).
// CN: 交集 (3)。
val i = set1.intersect(set2)

// RU: Разность (1, 2). Убрали то, что есть в set2.
// CN: 差集 (1, 2)。移除了 set2 中存在的元素。
val s = set1.subtract(set2) // или set1 - set2
```

---

3. Карты (Maps)
映射 (Maps)
Карта (или словарь) хранит пары "Ключ-Значение". Ключи уникальны. 映射（或字典）存储"键-值"对。键是唯一的。

---

4. Получение ключей и значений
获取键和值
У карты есть свойства `keys` (возвращает Set ключей) и `values` (возвращает Collection значений). Map 拥有 `keys` 属性（返回键的 Set）和 `values` 属性（返回值的 Collection）。

```
val map = mapOf("A" to 1, "B" to 2)

// RU: Все ключи ["A", "B"].
// CN: 所有键 ["A", "B"]。
val allKeys = map.keys

// RU: Все значения [1, 2].
// CN: 所有值 [1, 2]。
val allValues = map.values
```

---

5. Хильтрация (Filtering Maps)
过滤
Можно фильтровать карту по ключам, значениям или по самим записям (entries). 可以按键、值或条目 (entries) 本身过滤 Map。

```
val map = mapOf("Key1" to 10, "Key2" to 20)

// RU: Оставить ключи, начинающиеся на "K".
// CN: 保留以 "K" 开头的键。
val fKeys = map.filterKeys { it.startsWith("K") }

// RU: Оставить значения больше 15.
// CN: 保留大于 15 的值。
val fValues = map.filterValues { it > 15 }
```

---

6. Операции + и - (Map Operators)
+ 和 - 操作符
Эти операторы создают новую карту, добавляя или удаляя записи (не меняя исходную карту, даже если она read-only). 这些操作符创建新 Map，添加或删除条目（不更改原始 Map，即使它是只读的）。

```
val map = mapOf("A" to 1)

// RU: Новая карта {A=1, B=2}.
// CN: 新 Map {A=1, B=2}。
val newMap = map + ("B" to 2)

// RU: Новая карта без ключа "A".
// CN: 不包含键 "A" 的新 Map。
val lessMap = newMap - "A"
```

---

7. Добавление и обновление
添加和更新
Требуется `MutableMap`. Используется метод `put` или индексатор `[]`. 需要 `MutableMap`。使用 `put` 方法或索引器 `[]`。

```
val mutableMap = mutableMapOf("A" to 1)

// RU: Добавление новой пары.
// CN: 添加新对。
mutableMap["B"] = 2  // mutableMap.put("B", 2)

// RU: Обновление значения для существующего ключа.
// CN: 更新现有键的值。
mutableMap["A"] = 100
```

---

8. Удаление
删除
Требуется `MutableMap`. Используется метод `remove(key)`. 需要 `MutableMap`。使用 `remove(key)` 方法。

```
val mutableMap = mutableMapOf("A" to 1, "B" to 2)

// RU: Удаляет запись по ключу "A".
// CN: 按键 "A" 删除条目。
```

```
mutableMap.remove("A")

// RU: Также можно через оператор -=.
// CN: 也可以通过 -= 操作符。
mutableMap -= "B"
```

17. Последовательности (Sequence). Отличие коллекций и последовательностей. Порядок обработки элементов в коллекции. Порядок обработки элементов в последовательности. Последовательность. Промежуточная и завершающая операции. Конструирование последовательностей из элементов и итерируемых. Конструирование последовательностей из функций.

Это 17-й вопрос (Question 17), посвященный последовательностям (`Sequence`) в Kotlin. Это ключевая тема для оптимизации производительности при работе с большими наборами данных.
这是 第 17 题 (Question 17)，关于 Kotlin 中的序列 (`Sequence`)。这是优化大数据集处理性能的关键主题。
Мы разберем каждый пункт по порядку. (Без картинок, как вы просили). 我们将按顺序解析每一点。（按照您的要求，不含图片）。

1. Последовательности (Sequence)

序列

`Sequence` — это контейнер, который (в отличие от `List`) не хранит элементы в памяти, а вычисляет их по одному во время итерации. `Sequence` 是一个容器，与 `List` 不同，它不将元素存储在内存中，而是在迭代期间逐个计算它们。

2. Отличие коллекций и последовательностей

集合与序列的区别

- Коллекции (Iterable): “Жадные” (Eager). Выполняют каждый шаг обработки сразу для всех элементов и создают промежуточные списки.
- Последовательности (Sequence): “Ленивые” (Lazy). Вычисляют элементы только тогда, когда они действительно нужны (в самом конце).
- 集合 (Iterable): “急切” (Eager)。立即对所有元素执行每个处理步骤，并创建中间列表。
- 序列 (Sequence): “懒惰” (Lazy)。仅当真正需要元素时（在最后）才计算它们。

3. Порядок обработки элементов в коллекции

集合中元素的处理顺序

Обработка идет “горизонтально”. Сначала завершается операция A для всего списка, затем операция B для всего списка. 处理是“水平”进行的。首先完成整个列表的操作 A，然后完成整个列表的操作 B。

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

// RU: Сначала map создаст новый список [2, 4, 6].
// CN: 首先 map 创建一个新列表 [2, 4, 6]。
// RU: Затем filter создаст еще один список [4, 6].
// CN: 然后 filter 创建另一个列表 [4, 6]。
list.map { it * 2 }
    .filter { it > 2 }
```

4. Порядок обработки элементов в последовательности

序列中元素的处理顺序

Обработка идет “вертикально”. Первый элемент проходит через цепочку операций (A -> B), затем второй элемент (A -> B) и т.д. Промежуточные коллекции не создаются. 处理是“垂直”进行的。第一个元素通过操作链 (A -> B)，然后是第二个元素 (A -> B)，依此类推。不创建中间集合。

```
// RU: Элемент 1 -> map -> filter.
// CN: 元素 1 -> map -> filter.
// RU: Элемент 2 -> map -> filter.
// CN: 元素 2 -> map -> filter.
sequence.map { it * 2 }.filter { it > 2 }
```

5. Последовательность. Промежуточная и завершающая операции.

序列。中间操作和末端操作

Операции в `Sequence` делятся на два типа:

1. Промежуточные (Intermediate): Возвращают новую `Sequence` (например, `map`, `filter`). Они не выполняются сразу.
2. Завершающие (Terminal): Возвращают результат (List, Int, Unit). Они запускают выполнение всей цепочки.

`Sequence` 中的操作分为两种类型：

1. 中间操作 (Intermediate): 返回新的 `Sequence`（例如 `map`, `filter`）。它们不会立即执行。
2. 末端操作 (Terminal): 返回结果 (List, Int, Unit)。它们触发整个链的执行。

```
val seq = sequenceOf(1, 2, 3)

// RU: Ничего не происходит (лениво).
// CN: 什么也没发生（懒惰）。
val mapped = seq.map { println(it); it * 2 }

// RU: toList() — терминальная операция. Только сейчас map запустится.
// CN: toList() 是末端操作。只有现在 map 才会运行。
val result = mapped.toList()
```

6. Конструирование последовательностей из элементов и итерируемых

从元素和可迭代对象构建序列

Самый простой способ создать последовательность. 创建序列最简单的方法。

```
// 1. Из элементов (From elements)
val s1 = sequenceOf(1, 2, 3, 4)

// 2. Из коллекции (From collection)
val list = listOf("a", "b", "c")
// RU: Превращаем "тяжелый" список в последовательность.
// CN: 将“重”列表转换为序列。
val s2 = list.asSequence()
```

7. Конструирование последовательностей из функций

从函数构建序列

Позволяет создавать бесконечные последовательности с помощью `generateSequence`. 允许使用 `generateSequence` 创建无限序列。

```
// RU: Первый элемент 1. Каждый следующий: предыдущий + 1.
// CN: 第一个元素是 1。每一个后续元素：前一个 + 1。
val naturalNumbers = generateSequence(1) { it + 1 }

// RU: Берем первые 5 элементов (иначе будет бесконечный цикл).
// CN: 取前 5 个元素（否则会是无限循环）。
val firstFive = naturalNumbers.take(5).toList() // [1, 2, 3, 4, 5]
```

18. Делегирование класса. Переопределение члена интерфейса, реализованного делегированием. Доступ к реализациям членов интерфейса. Делегирование свойств. Сигнатура делегата. Действия делегата. Ленивые свойства. Обозреваемые свойства. Франение свойств в ассоциативном списке.

Это 18-й вопрос (Question 18), посвященный мощному механизму делегирования в Kotlin. Это реализация паттерна “Delegation” на уровне языка.
这是 第 18 题 (Question 18)，关于 Kotlin 中强大的委托机制。这是语言层面对“委托”模式的实现。
Мы разберем каждый пункт подробно (без картинок). 我们将详细解析每一点（不含图片）。

1. Делегирование класса

类委托

Kotlin позволяет передать реализацию интерфейса другому объекту с помощью ключевого слова `by`. Это альтернатива наследованию. Kotlin 允许使用关键字 `by` 将接口的实现委托给另一个对象。这是继承的替代方案。

```
interface Base {
    fun print()
}

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

// RU: Derived делегирует реализацию интерфейса Base объекту b.
// CN: Derived 将 Base 接口的实现委托给对象 b。
class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print() // Output: 10
}
```

2. Переопределение члена интерфейса, реализованного делегированием

重写通过委托实现的接口成员

Даже если класс делегирует реализацию, он может переопределить методы самостоятельно. Собственная реализация имеет приоритет. 即使类委托了实现，它也可以自己重写方法。自己的实现具有优先级。

```
class Derived(b: Base) : Base by b {
    // RU: Переопределяем метод. Делегат (b.print) вызван не будет.
    // CN: 重写方法。委托 (b.print) 不会被调用。
    override fun print() {
        println("Override")
    }
}
```

3. Доступ к реализациям членов интерфейса

访问接口成员的实现

Если вы переопределили метод, вы не можете вызвать реализацию делегата через `super`. Чтобы иметь к ней доступ, нужно сохранить делегат в свойстве класса. 如果您重写了方法，您不能通过 `super` 调用委托的实现。要访问它，必须将委托保存在类属性中。

```
// RU: Сохраняем 'b' как private val, чтобы обращаться к нему.
// CN: 将 'b' 保存为 private val，以便访问它。
class Derived(private val b: Base) : Base by b {
    override fun print() {
        // super.print() // Error!
        b.print() // OK
        println("Additional logic")
    }
}
```

4. Делегирование свойств

属性委托

Позволяет перенести логику геттеров и сеттеров в отдельный класс. Синтаксис: `val/var <имя> by <делегат>`. 允许将 Getter 和 Setter 的逻辑移至单独的类中。语法：`val/var <名称> by <委托>`。

```
class Example {
    // RU: Логика чтения/записи 'p' находится в классе Delegate.
    // CN: 'p' 的读/写逻辑位于 Delegate 类中。
    var p: String by Delegate()
}
```

5. Сигнатура делегата

委托签名

Класс-делегат должен иметь методы `getValue` (и `setValue` для var) с оператором `operator`. 委托类必须具有带有 `operator` 操作符的 `getValue` 方法（对于 var 还需要 `setValue`）。

```kotlin
import kotlin.reflect.KProperty

class Delegate {
    // RU: Метод для чтения (get).
    // CN: 读取方法 (get)。
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return "Delegate value"
    }

    // RU: Метод для записи (set) — нужен только для var.
    // CN: 写入方法 (set) — 仅 var 需要。
    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: Str
        ing) {
        println("Assigned: $value")
    }
}
```

6. Действия делегата

委托操作

Внутри методов `getValue` и `setValue` можно выполнять любые действия: логирование, проверку данных, чтение из базы и т.д. 在 `getValue` 和 `setValue` 方法内部，可以执行任何操作：记录日志、数据验证、从数据库读取等。

7. Ленивые свойства (Lazy Properties)

延迟属性 (Lazy Properties)

Стандартный делегат `lazy`. Значение вычисляется только при первом обращении и запоминается. 标准委托 `lazy`。值仅在第一次访问时计算并被记住。

```kotlin
// RU: Вычисление произойдет только при первом вызове lazyValue.
// CN: 只有在第一次调用 lazyValue 时才会进行计算。
val lazyValue: String by lazy {
    println("Computed!")
    "Hello"
}
```

8. Обозреваемые свойства (Observable Properties)

可观察属性

Стандартный делегат `Delegates.observable`. Позволяет выполнить код при каждом изменении свойства. 标准委托 `Delegates.observable`。允许在每次属性更改时执行代码。

```kotlin
import kotlin.properties.Delegates

class User {
    // RU: Лямбда вызывается после каждого присваивания.
    // CN: 每次赋值后都会调用 Lambda。
    var name: String by Delegates.observable("No Name") { prop, old, new ->
        println("$old -> $new")
    }
}
```

9. Франение свойств в ассоциативном списке (Storing properties in a Map)

在映射 (Map) 中存储属性

Можно использовать `Map` как делегат. Это удобно для парсинга JSON или динамических конфигураций. 可以使用 `Map` 作为委托。这对解析 JSON 或动态配置非常方便。

```kotlin
class User(val map: Map<String, Any?>) {
    // RU: Значение берется из карты по ключу "name".
    // CN: 值从 Map 中按键 "name" 获取。
    val name: String by map
    val age: Int    by map
}

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

19. Внутренний DSL. Синтаксические особенности языка Kotlin для создания DSL. Хункция-расширение. Инфиксный вызов функции. Перегрузка операторов. Соглашение о методе get/set. Лямбда-выражение вне круглых скобок. Псевдонимы типа. Мульти-декларации. HTML-построитель. JSON-построитель.

Это 19-й вопрос (Question 19), посвященный созданию DSL (Domain Specific Languages) в Kotlin. Это одна из самых продвинутых и интересных тем языка.
这是 第 19 题 (Question 19)，关于在 Kotlin 中创建 DSL（领域特定语言）。这是该语言最高级和最有趣的主题之一。
Мы разберем каждый пункт подробно. 我们将详细解析每一点。

1. Внутренний DSL (Internal DSL)

内部 DSL

Внутренний DSL — это язык, созданный с использованием синтаксиса основного языка (Kotlin) для решения специфических задач (например, построение HTML, конфигурация Gradle, SQL-запросы). 内部 DSL 是使用主语言 (Kotlin) 的语法创建的语言，用于解决特定任务（例如构建 HTML、Gradle 配置、SQL 查询）。

2. Синтаксические особенности языка Kotlin для создания DSL

用于创建 DSL 的 Kotlin 语法特性

Kotlin обладает набором функций, которые позволяют убрать "синтаксический шум" (скобки, точки, запятые), делая код похожим на естественный язык или структуру данных. Kotlin 拥有一套特性，允许消除"语法噪音"（括号、点、逗号），使代码看起来像自然语言或数据结构。

3. Хункция-расширение (Extension Function)

扩展函数

Позволяет добавлять методы в существующие классы. В DSL это используется для добавления функций-тегов (например, `body`, `div`) в контекст построителя. 允许向现有类添加方法。在 DSL 中，这用于向构建器上下文中添加标签函数（例如 `body`, `div`）。

```kotlin
// RU: Добавляем функцию div в класс HTML.
// CN: 向 HTML 类添加 div 函数。
fun HTML.div(block: DIV.() -> Unit) { ... }
```

4. Инфиксный вызов функции (Infix Call)

中缀函数调用

Позволяет вызывать функцию без точки и круглых скобок, если у нее один аргумент. Это делает код похожим на предложение. 如果函数只有一个参数，则允许在没有点和圆括号的情况下调用它。这使代码看起来像一个句子。

```kotlin
infix fun String.shouldBe(expected: String) { /*...*/ }

// RU: Вызов как в английском языке.
// CN: 像英语一样调用。
"Result" shouldBe "Result"
```

5. Перегрузка операторов (Operator Overloading)

操作符重载

Позволяет использовать знаки `+`, `-`, `*` для своих типов. В DSL часто используется унарный плюс `+` для добавления строк или оператор `invoke ()`. 允许对自己定义的类型使用符号 `+`, `-`, `*`。在 DSL 中，常使用一元加号 `+` 来添加字符串，或使用 `invoke ()` 操作符。

```kotlin
operator fun String.unaryPlus() {
    // RU: Логика добавления строки в тег.
    // CN: 将字符串添加到标签的逻辑。
}

html {
    +"This is text inside tag"
}
```

6. Соглашение о методе get/set

get/set 方法约定

Если определить операторы `get` и `set`, можно использовать квадратные скобки `[]`. Это удобно для настройки маппингов или доступа к элементам. 如果定义了 `get` 和 `set` 操作符，则可以使用方括号 `[]`。这对配置映射或访问元素非常方便。

```kotlin
// DSL для конфигурации
config["port"] = 8080 // Вызывает config.set("port", 8080)
```

7. Лямбда-выражение вне круглых скобок (Trailing Lambda)

圆括号外的 Lambda 表达式

Самая важная фича для DSL. Если лямбда — последний аргумент, она пишется `{ }` вместо `({ })`. Это позволяет создавать вложенные структуры. DSL 最重要的特性。如果 Lambda 是最后一个参数，则写成 `{ }` 而不是 `({ })`。这允许创建嵌套结构。

```kotlin
// RU: Вместо html({ body({ ... }) })
// CN: 代替 html({ body({ ... }) })
html {
    body {
        div { ... }
    }
}
```

8. Псевдонимы типа (Type Aliases)

类型别名

Позволяют дать понятное имя функциональному типу или сложному дженерику, улучшая читаемость DSL. 允许给函数类型或复杂的泛型起一个易懂的名字，提高 DSL 的可读性。

```kotlin
typealias ElementInit = Element.() -> Unit

fun body(init: ElementInit) { ... }
```

9. Мульти-декларации (Destructuring Declarations)

多重声明（解构声明）

Позволяют разложить объект на переменные. В DSL используется в циклах по картам. 允许将对象分解为变量。在 DSL 中用于遍历 Map。

```kotlin
val map = mapOf(1 to "A", 2 to "B")

// RU: Разлагаем запись карты на ключ и значение.
// CN: 将 Map 条目分解为键和值。
for ((key, value) in map) {
    println("$key -> $value")
}
```

10. HTML-построитель (HTML Builder)

HTML 构建器

Классический пример Kotlin DSL. Это типобезопасный способ генерации HTML-кода (библиотека `kotlinx.html`). Kotlin DSL 的经典示例。这是生成 HTML 代码的类型安全方式（`kotlinx.html` 库）。

```kotlin
val myHtml = html {
    body {
        div {
            a(href = "https://kotlinlang.org") { +"Kotlin" }
        }
    }
}
```

11. JSON-построитель (JSON Builder)

JSON 构建器

Аналогичный подход для создания JSON-структур. Выглядит чище, чем создание строк вручную. 创建 JSON 结构的类似方法。看起来比手动创建字符串更干净。

```
val json = json {
    "name" to "John"
    "age" to 30
    "contacts" to jsonArray {
        +"email@example.com"
        +"phone number"
    }
}
```

20. Совместное использование языков Kotlin и Java. Вызов кода Java из Kotlin. Геттеры и сеттеры. Экранирование идентификаторов Java. Нулевые ссылки из Java. Аннотации допустимости null значений. Родовые типы Java в Kotlin. Массивы Java в Kotlin. Методы Java с переменным числом параметров.

Это 20-й вопрос (Question 20), посвященный совместимости Kotlin и Java (Interoperability). Это одна из главных причин успеха Kotlin — вы можете использовать старый Java-код в новом проекте на Kotlin без проблем.
这是 第 20 题 (Question 20)，关于 Kotlin 和 Java 的互操作性 (Interoperability)。这是 Kotlin 成功的主要原因之一 —— 您可以在新的 Kotlin 项目中毫无问题地使用旧的 Java 代码。
Мы разберем каждый пункт подробно. 我们将详细解析每一点。

1. Совместное использование языков Kotlin и Java

Kotlin 和 Java 的共同使用

Kotlin спроектирован так, чтобы быть на 100% совместимым с Java. Они оба компилируются в байт-код JVM. Вы можете иметь файлы `.java` и `.kt` в одном проекте. Kotlin 被设计为与 Java 100% 兼容。它们都编译为 JVM 字节码。您可以在同一个项目中同时拥有 `.java` 和 `.kt` 文件。

2. Вызов кода Java из Kotlin

从 Kotlin 调用 Java 代码

Классы Java можно создавать и использовать так же, как классы Kotlin. `new` писать не нужно. 可以像 Kotlin 类一样创建和使用 Java 类。不需要写 `new`。

```
// Java: public class JavaUser { ... }

// Kotlin:
// RU: Создаем экземпляр Java-класса.
// CN: 创建 Java 类的实例。
val user = java.util.ArrayList<String>()
user.add("Item")
```

3. Геттеры и сеттеры (Getters and Setters)

Getter 和 Setter

Методы Java, следующие конвенции JavaBeans (`getSomething`, `setSomething`), в Kotlin видны как свойства. 遵循 JavaBeans 约定 (`getSomething`, `setSomething`) 的 Java 方法在 Kotlin 中被视为属性。

```
/* Java:
   public String getName() { ... }
   public void setName(String name) { ... }
*/

// Kotlin:
// RU: Вместо user.getName()
// CN: 代替 user.getName()
println(user.name)

// RU: Вместо user.setName("Alice")
// CN: 代替 user.setName("Alice")
user.name = "Alice"
```

4. Экранирование идентификаторов Java

Java 标识符转义

Если имя метода в Java совпадает с ключевым словом в Kotlin (например, `in`, `is`, `object`), его нужно заключить в обратные кавычки
`. 如果 Java 中的方法名与 Kotlin 中的关键字（例如 `in`, `is`, `object`）冲突，则需要将其包含在反引号 中。

```
// Java: public void is() { ... }

// Kotlin:
// RU: Экранируем имя метода.
// CN: 转义方法名。
javaObject.`is`()
```

5. Нулевые ссылки из Java (Platform Types)

来自 Java 的空引用（平台类型）

Если в Java типе нет аннотаций (`@Nullable`/`@NotNull`), Kotlin не знает, может ли он быть null. Такой тип называется Платформенным типом (`String!`). 如果 Java 类型没有注解 (`@Nullable`/`@NotNull`)，Kotlin 不知道它是否可以是 null。这种类型被称为平台类型 (`String!`)。

- Вы можете работать с ним как с `String` (риск `NullPointerException`).
- Вы можете работать с ним как с `String?` (безопасно).
- 您可以将其视为 `String` 处理（有 `NullPointerException` 风险）。
- 您可以将其视为 `String?` 处理（安全）。

6. Аннотации допустимости null значений

空值允许注解

Kotlin понимает стандартные аннотации Java (из `javax.annotation`, `org.jetbrains.annotations`, `android.support.annotation` и т.д.). Kotlin 理解标准的 Java 注解（来自 `javax.annotation`, `org.jetbrains.annotations`, `android.support.annotation` 等）。

- Java `@Nullable String` -> Kotlin `String?`
- Java `@NotNull String` -> Kotlin `String`

7. Родовые типы Java в Kotlin

Kotlin 中的 Java 泛型

Генерики Java преобразуются в генерики Kotlin. Java Wildcards (`? extends`) преобразуются в `out` и `in` (site-variance). Java 泛型转换为 Kotlin 泛型。Java 通配符 (`? extends`) 转换为 `out` 和 `in` (site-variance)。

- `List<? extends String>` -> `List<out String>`
- `List<? super String>` -> `List<in String>`
- `List` (raw type) -> `List<*>`!

8. Массивы Java в Kotlin

Kotlin 中的 Java 数组

Массивы в Kotlin инвариантны (в отличие от Java). Примитивные массивы Java (`int[]`) отображаются в специальные классы (`IntArray`). Kotlin 中的数组是不变的（与 Java 不同）。Java 的原始数组 (`int[]`) 映射到特殊的类 (`IntArray`)。

- Java `int[]` -> Kotlin `IntArray`
- Java `String[]` -> Kotlin `Array<String>`

```
// Java: public void process(int[] data);

// Kotlin:
val data = IntArray(5)
javaObj.process(data)
```

9. Методы Java с переменным числом параметров (Varargs)

带可变参数的 Java 方法

Чтобы передать массив в Java-метод, принимающий `varargs` (`...`), в Kotlin нужно использовать оператор разыменования (spread operator) `*`. 要将数组传递给接收 `varargs` (`...`) 的 Java 方法，需要在 Kotlin 中使用展开操作符 `*`。

```
// Java: void log(String... lines);

val lines = arrayOf("Line 1", "Line 2")

// RU: Звездочка обязательна!
// CN: 星号是必须的！
javaLog.log(*lines)
```

21. Совместное использование языков Kotlin и Java. Вызов кода Kotlin из Java. Свойства. Хункции уровня пакета. Смена имени генерируемого Java класса. Поля экземпляра. Статические поля. Статические методы.

Это 21-й вопрос (Question 21), посвященный обратной стороне совместимости: как Java видит код, написанный на Kotlin.
这是 第 21 题 (Question 21)，关于互操作性的另一方面：Java 如何看待用 Kotlin 编写的代码。
Мы разберем каждый пункт подробно без картинок. 我们将详细解析每一点，不使用图片。

1. Совместное использование языков Kotlin и Java. Вызов кода Kotlin из Java.

Kotlin 和 Java 的共同使用。从 Java 调用 Kotlin 代码。

Kotlin компилируется в стандартный байт-код JVM. Для Java классы Kotlin выглядят как обычные Java-классы, но есть нюансы с именованием и статикой. Kotlin 编译为标准的 JVM 字节码。对于 Java 而言，Kotlin 类看起来像普通的 Java 类，但在命名和静态方面存在细微差别。

2. Свойства (Properties)

属性

Свойства Kotlin (`val`/`var`) компилируются в приватное поле + геттер/сеттер. Kotlin 属性 (`val`/`var`) 编译为私有字段 + Getter/Setter。

- `val x: Int` -> `int getX()`
- `var y: Int` -> `int getY()`, `void setY(int value)`
- `isBool: Boolean` -> `boolean isBool()`, `void setBool(boolean value)`

```
// Java code:
MyClass obj = new MyClass();

// RU: Вызываем геттер, который сгенерировал Kotlin.
// CN: 调用 Kotlin 生成的 Getter。
String name = obj.getName();
```

3. Хункции уровня пакета (Package-level functions)

包级函数

Хункции, объявленные в файле вне класса (top-level), компилируются в статические методы класса с именем файла + суффикс `Kt`. 在类外部（顶层）声明的函数编译为文件名为名 + 后缀 `Kt` 的类的静态方法。

- Хайл `Utils.kt` -> Класс `UtilsKt`

```
// Utils.kt
fun doWork() {}
```

```
// Java code:
// RU: Обращаемся через имя файла + Kt.
// CN: 通过文件名 + Kt 访问。
UtilsKt.doWork();
```

4. Смена имени генерируемого Java класса

更改生成的 Java 类名

Если имя `UtilsKt` вам не нравится, его можно изменить с помощью аннотации `@file:JvmName`, размещенной в самом начале файла. 如果您不喜欢 `UtilsKt` 这个名字，可以使用放置在文件开头的注解 `@file:JvmName` 来更改它。

```
// Utils.kt
@file:JvmName("Helper")
package com.example

fun doWork() {}
```

```java
// Java code:
// RU: Теперь имя класса чистое.
// CN: 现在类名很干净。
Helper.doWork();
```

5. Поля экземпляра (Instance Fields)

实例字段

Если вам нужно, чтобы свойство Kotlin было обычным публичным полем Java (без геттеров/сеттеров), используйте аннотацию `@JvmField`. 如果您需要 Kotlin 属性成为普通的 Java 公共字段（无 Getter/Setter），请使用 `@JvmField` 注解。

```kotlin
class User {
    @JvmField
    var id: Int = 1
}
```

```java
// Java code:
User u = new User();
// RU: Прямой доступ к полю, как в Java (u.id).
// CN:像 Java 一样直接访问字段 (u.id)。
u.id = 2;
```

6. Статические поля (Static Fields)

静态字段

В Kotlin нет `static`. Статические поля создаются двумя способами:

1. `const val` (в `companion object` или top-level): становится `public static final`.
2. `@JvmField` внутри `object`: становится `public static`.

Kotlin 中没有 `static`。静态字段通过两种方式创建：

1. `const val`（在 `companion object` 或顶层中）：变为 `public static final`。
2. `object` 内部的 `@JvmField`：变为 `public static`。

```kotlin
object Config {
    const val VERSION = 1
    @JvmField val URL = "http"
}
```

```java
// Java:
int v = Config.VERSION;
String u = Config.URL;
```

7. Статические методы (Static Methods)

静态方法

Методы внутри `object` или `companion object` по умолчанию не являются статическими в Java (они вызываются через `INSTANCE`). Чтобы сделать их настоящими `static` методами, нужна аннотация `@JvmStatic`. `object` 或 `companion object` 内部的方法在 Java 中默认不是静态的（它们通过 `INSTANCE` 调用）。要使它们成为真正的 `static` 方法，需要 `@JvmStatic` 注解。

```kotlin
class KeyGenerator {
    companion object {
        @JvmStatic
        fun generate(): String = "Key"
    }
}
```

```java
// Java:
// RU: Работает как обычный статический метод.
// CN: 像普通的静态方法一样工作。
String key = KeyGenerator.generate();

// RU: Без @JvmStatic пришлось бы писать: KeyGenerator.Companion.generate()
// CN: 没有 @JvmStatic 则必须写成：KeyGenerator.Companion.generate()
```