

## 13.ВВЕДЕНИЕ В KOTLIN

### 13.1 Введение

Kotlin – это современный язык программирования, работающий на виртуальной машине Java (JVM). Данный язык имеет простой для восприятия синтаксис и поддержку всех существующих Java-библиотек. В 2017 году на конференции Google I/O Android-сообщество анонсировало, что Kotlin станет официальным языком программирования для данной платформы.

Ресурсы

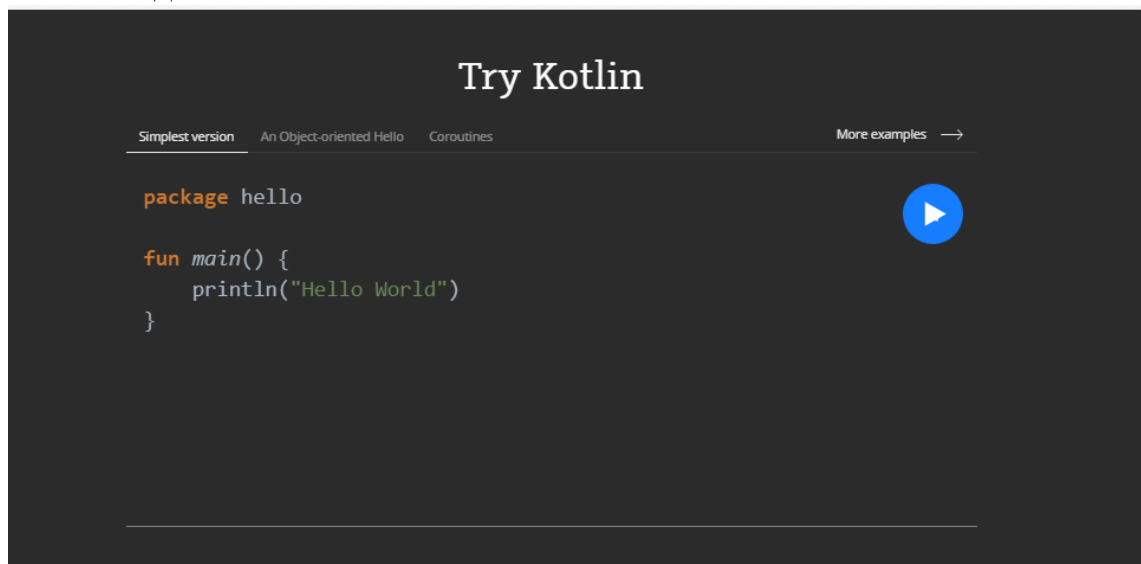
<https://kotlinlang.ru/>

<https://kotlinlang.org/>

<https://try.kotlinlang.org/>

Официальный сайт данного проекта. В разделе Docs <https://kotlinlang.org/docs/reference/> вы сможете найти подробную документацию, которая охватывает все основные фичи и концепты данного языка. Раздел туториалы <https://kotlinlang.org/docs/tutorials/> содержит различные практические пошаговые уроки о том, как настроить рабочую среду и, как работать с компилятором.

Также сайт содержит редактор <https://try.kotlinlang.org/> для работы с Kotlin, который является веб-приложением, позволяющее вам попробовать поработать с данным языком.

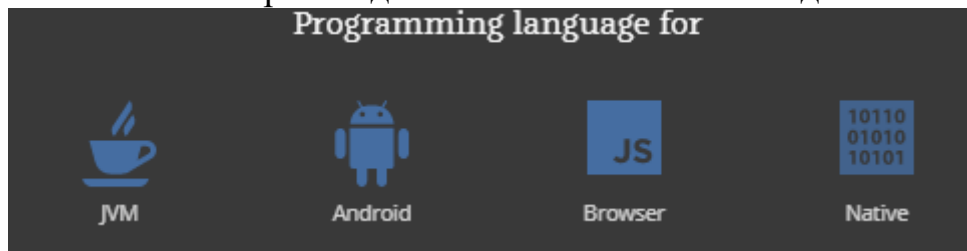


<https://android.jlelse.eu/learn-kotlin-while-developing-an-android-app-introduction-567e21ff9664>

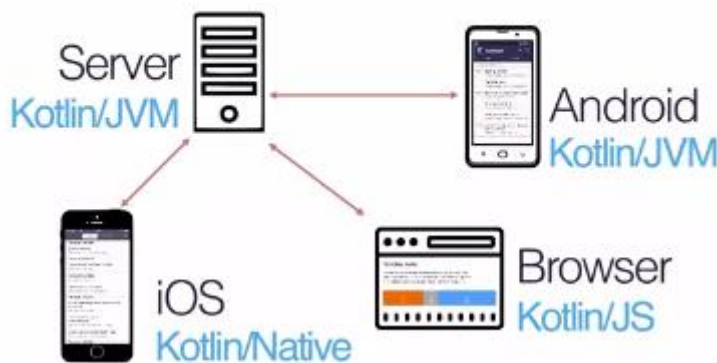
Итак, Kotlin - современный мультиплатформенный статически типизированный объектно-ориентированный язык программирования. Являясь полностью совместимым с Java, Kotlin предоставляет дополнительные возможности, которые призваны упростить повседневную работу программиста и повысить его продуктивность.

Как и Java, Kotlin является статически типизированным языком. Однако в Kotlin вы можете опускать типы, и он часто выглядит таким же лаконичным, как и некоторые другие динамически типизированные языки. Kotlin безопасен и даже безопаснее Java с точки зрения того, что компилятор Kotlin может помочь предотвратить еще больше возможных типов ошибок. Одной из основных характеристик Kotlin является его хорошая совместимость с Java. Сейчас трудно назвать Java современным языком. Тем не менее, он имеет настолько огромную экосистему, что было бы действительно трудно воссоздать его с нуля, если вы используете новый язык.

Kotlin - это не только JVM или Android, есть и другие целевые платформы. Конечно, JVM - это первое, но вы также можете скомпилировать код Kotlin в JavaScript или даже сейчас в нативный код.



## Multiplatform projects



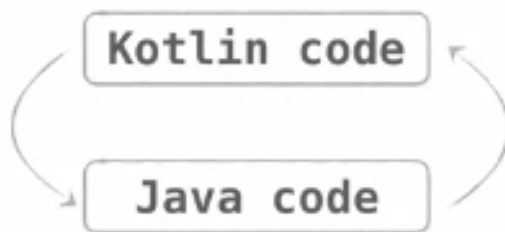
Ваш исходный код на Kotlin будет компилироваться, или транслироваться, в байт-код Java и выполняться под управлением JVM



Kotlin позволяет программировать в функциональном стиле, но не требует этого.

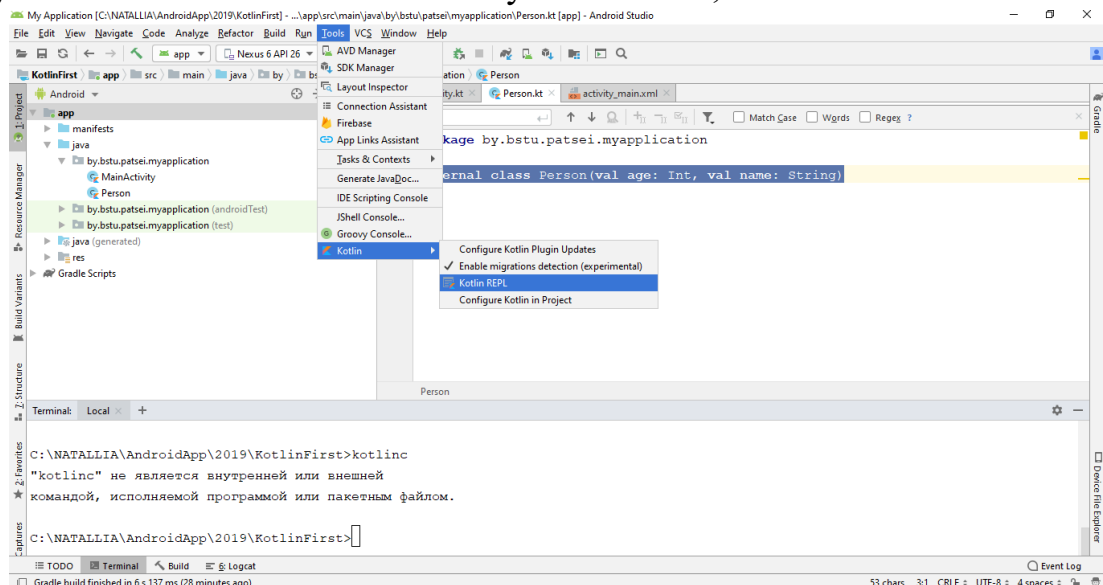
В программном коде на Kotlin вы можете совмещать объектно-ориентированный и функциональный подходы, используя для каждой решаемой проблемы наиболее подходящий инструмент.

Начать использовать Kotlin легко, особенно если вы Java-разработчик. Вы можете использовать все существующие фреймворки и библиотеки Java от Kotlin. Вы даже можете смешать код Kotlin и Java в одном проекте. Kotlin - это язык JVM, поэтому вы можете легко вызывать код Java из Kotlin, но это также работает в другом направлении. Kotlin разработан таким образом, чтобы использовать все его функции из Java было просто.



### *Интерактивная оболочка*

Для быстрого опробования небольших фрагментов кода на Kotlin можно использовать интерактивную оболочку (так называемый цикл REPL - Read Eval Print Loop: чтение ввода, выполнение, вывод результата, повтор). В REPL можно вводить код на Kotlin строку за строкой и сразу же видеть результаты выполнения. Чтобы запустить REPL, выполните



## 13.2 Основные элементы: переменные и функции

Рассмотрим следующую функцию

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

- Объявления функций начинаются с ключевого слова `fun`.
- Тип параметра указывается после его имени. Это относится и к объявлениям переменных.
- Функцию можно объявить на верхнем уровне в файле - её не обязательно помещать в класс.
- Массивы - это просто классы.
- Вместо `System.out.println` можно писать просто `println`. Стандартная библиотека Kotlin включает множество оберток с лаконичным синтаксисом для функций в стандартной библиотеке Java, и `println` - одна из них.
- Точку с запятой в конце строки можно опустить, как и во многих других современных языках.

Поменяем функцию

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

Тип возвращаемого значения указывается после списка параметров и отделяется от него двоеточием.

Оператор *if* является выражением, возвращающим значение. Это похоже на тернарный оператор в Java: `(a > b) ? a : b`.

В языке Kotlin оператор *if* - это выражение, а не инструкция. Разница между выражениями и инструкциями состоит в том, что выражение имеет значение, которое можно использовать в других выражениях, в то время как инструкции всегда являются элементами верхнего уровня в охватывающем блоке и не имеют собственного значения. В **Java** все управляющие структуры - **инструкции**. В **Kotlin** большинство управляющих структур, кроме циклов (*for*, *do u do/while*), - **выражения**.

Поменяем функцию

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

Её тело состоит из единственного выражения.

Если тело функции заключено в фигурные скобки, мы говорим, что такая функция имеет тело-блок (*block body*). Функция, возвращающая выражение напрямую, имеет тело-выражение (*expression body*).

Поменяем функцию

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

Опустить тип возвращаемого значения можно только в функциях с телом-выражением.

Каждая переменная и каждое выражение имеют тип, и каждая функция имеет тип возвращаемого значения. Но для функций с телом-выражением компилятор может проанализировать выражение и использовать его тип в качестве типа возвращаемого значения функции, даже когда он не указан явно.

### *Переменные*

В Java объявление переменной начинается с типа. Такой способ не поддерживается в Kotlin, объявление начинается с ключевого слова, а тип можно указать (или не указывать) после имени переменной.

```
val name = "Anna"  
val age = 22
```

или

```
val name :String = "Anna"  
val age : Int = 22
```

Так же, как в функциях с телом-выражением, если тип не указан явно, компилятор проанализирует инициализирующее выражение и присвоит его тип переменной. Если в объявлении переменной отсутствует инициализирующее выражение, её тип нужно указать явно.

Есть два ключевых слова для объявления переменной:

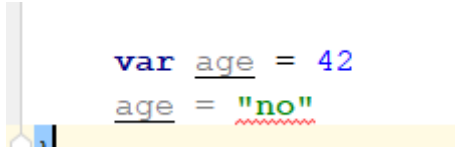
- **val (от value) - неизменяемая ссылка.** Переменной, объявленной с ключевым словом `val`, нельзя присвоить значение после инициализации. Такие переменные соответствуют финальным переменным в Java.
- **var (от variable) - изменяемая ссылка.** Значение такой переменной можно изменить. Такое объявление соответствует обычной (не финальной) переменной в Java.

По умолчанию нужно стремиться объявлять все переменные в Kotlin с ключевым словом `val`. Заменяйте его на `var` только при необходимости. Использование неизменяемых ссылок и объектов, а также функций без побочных эффектов приблизит ваш код к функциональному стилю.

Несмотря на невозможность изменить ссылку `val`, объект, на который она указывает, может быть изменяемым:

```
fun main() {
    val fio = arrayListOf("Ivanov")
    fio.add(" Sergey")
}
```

`var` позволяет менять значение переменной, но её тип фиксирован:



```
var age = 42
age = "no"
```

Компилятор определяет тип переменной только по инициализирующему выражению и не принимает во внимание всех последующих операций присваивания.

Если вам нужно сохранить в переменной значение другого типа, вы должны преобразовать его вручную или привести к нужному типу.

Kotlin позволяет использовать в строковых литералах ссылки на локальные переменные, добавляя к ним в начало символ `$`.

```
fun main() {
    val fio = arrayListOf("Ivanov")
    fio.add(" Sergey")
    println("Hello, $fio!")
}
```

*Hello, [Ivanov, Sergey]!*

Также можно помещать двойные кавычки внутрь других двойных кавычек, пока они входят в состав выражения:

```
println("Hello, ${if (args.size > 0) args[0] else "User"}!")
```

*Hello, User!*

## Классы и свойства

Рассмотрим простой *JavaBean*-класс *Person*

```
class Person {
    private final int age;
    private final String name;

    Person(int age, String name) {
        this.age = age;
        this.name = name;
    }

    public int getAge() {
        return age;
    }

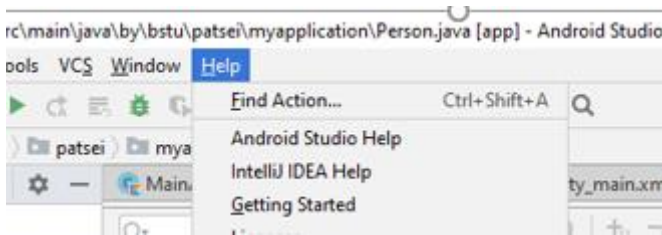
    public String getName() {
        return name;
    }
}
```

## Конвертер

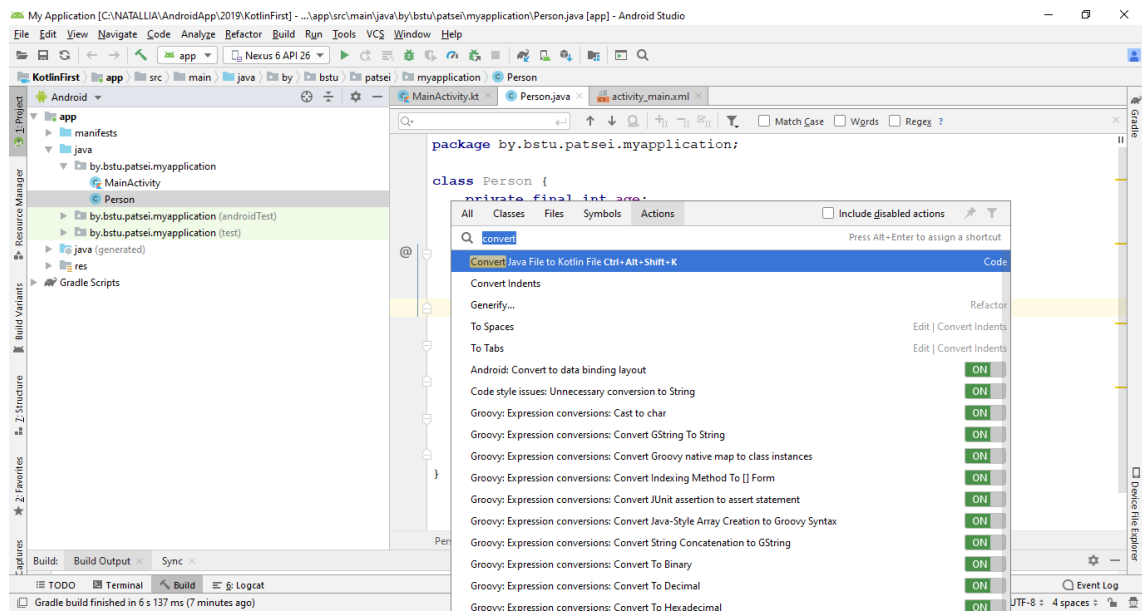
Существует мощный инструмент, который помогает ускорить процесс изучения языка на основе вашего опыта *Java*. Этот инструмент является автоматическим конвертером *Java* в *Kotlin*. Конвертер имеет два основных варианта использования. Изучение языка и включение его в существующее приложение *Java*. Когда вы знаете, как написать что-то на *Java*, но еще не знаете или не помните точную конструкцию на *Kotlin*, вы можете сначала написать ее на *Java*, а затем преобразовать.

Также чрезвычайно полезно использовать конвертер, когда вы начинаете добавлять *Kotlin* в существующее *Java*-приложение. Если у вас есть класс *Java* и вам нужно его изменить, вы можете предпочесть использование *Kotlin*. В этом случае вы можете сначала преобразовать этот класс *Java* в *Kotlin*, а затем вы сможете написать новый код на *Kotlin*.





Вы можете вызвать действие «Преобразовать файл Java в файл Kotlin» в IntelliJ IDEA или в студии Android. В результате он дает нам тот же код, написанный на Kotlin.



Вот что получится

```
internal class Person(val age: Int, val name: String)
```

Классы этого типа (содержащие только данные, без кода) часто называют *объектами-значениями* (*value objects*), и многие языки предлагают краткий синтаксис для их объявления.

Обратите внимание, что в ходе преобразования из *Java* в *Kotlin* пропал модификатор *public*. В *Kotlin* область видимости *public* принята по умолчанию, поэтому её можно не указывать.

Свойство в классе объявляется так же, как переменная: с помощью ключевых слов *val* и *var*. Свойство, объявленное как *val*, доступно только для чтения, а свойство *var* можно изменять.

```
internal class Person(var age: Int, val name: String)
```

Данный класс можно использовать и в *Java*, и в *Kotlin*, независимо от того, где он объявлен.

```
internal class Person(var age: Int, val name: String)

fun main(args: Array<String>) {
    var value = Person (23, "Bob");
    System.out.println(value.age);
}
```

Свойство *name*, объявленное на языке *Kotlin*, доступно *Java*-коду через метод доступа с именем *getName*.

## Пакеты и каталоги

В *Java* все классы находятся в пакетах. В *Kotlin* также существует понятие пакета, похожее на аналогичное понятие в *Java*. Каждый файл *Kotlin* может иметь инструкцию ***package*** в начале, и все объявления (классы, функции и свойства) в файле будут помещены в этот пакет. Объявления из других файлов в том же пакете можно использовать напрямую, а объявления из других пакетов нужно импортировать. Так же, как в *Java*, инструкции импорта помещаются в начало файла и начинаются с ключевого слова ***import***.

```
package by.bstu.patsei.myapplication

import java.util.Random

class Rectangle(val height: Int, val width: Int) {
    val isSquare: Boolean
    get() = height == width
}

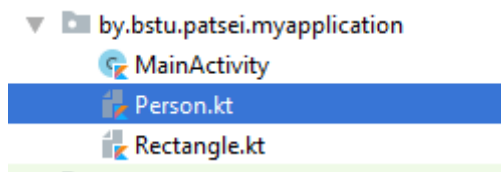
fun createRandomRectangle(): Rectangle {
    val random = Random()
    return Rectangle(random.nextInt(), random.nextInt())
}
```

*Kotlin* не делает различия между импортом классов и функций, что позволяет импортировать любые объявления с помощью ключевого слова *import*. Функции верхнего уровня можно импортировать по имени.

```
import by.bstu.patsei.myapplication.createRandomRectangle
```

Кроме того, можно импортировать все объявления из определенного пакета, добавив `*` после имени пакета. Обратите внимание, что такой импорт со звездочкой сделает видимыми не только классы, объявленные в пакете, но и свойства и функции верхнего уровня.

В большинстве случаев хорошим тоном считается следовать структуре каталогов в *Java* и организовывать исходные файлы в каталогах в соответствии со структурой пакета



## Константы времени компиляции

Константа времени компиляции объявляется вне какой-либо функции, даже не в пределах функции *main*, потому что ее значение присваивается во время компиляции (в момент, когда программа компилируется).

*main* и другие функции вызываются во время выполнения (когда программа запущена), и переменные внутри функций получают свои значения в этот период. Константа времени компиляции уже существует к этому моменту.

Константы времени компиляции могут иметь значения только одного из следующих базовых типов — а использование более сложных типов может сделать компиляцию невозможной. ***String Int Double Float Long Short Byte Char Boolean***

```
const val MAX_TIME: Int = 50

fun main(args: Array<String>) {
    var value = Person (23, "Bob");
    System.out.println(value.age);
}
```

## 13.3 Условные конструкции

### Условные выражения

Условное выражение — это почти условный оператор, с той лишь разницей, что результат оператора *if/else* присваивается переменной, которая будет использоваться в дальнейшем.

```
val color = if (MAX_TIME > 50) "GREEN" else "RED"
```

В *Kotlin* тернарный оператор отсутствует, поскольку, в отличие от *Java*, выражение *if* возвращает значение.

### Конструкция «when»

Её можно считать заменой конструкции *switch* в *Java*, но с более широкими возможностями и более частым применением на практике.

Добавим перечисление цветов.

```
enum class Color {  
    RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET  
}
```

В языке *Kotlin* *enum* - это так называемое «мягкое» ключевое слово (soft keyword): оно имеет особое значение только перед ключевым словом *class*, в других случаях его можно использовать как обычное имя.

Точно, как в *Java*, перечисления - это не просто списки значений: в классах перечислений можно объявлять свойства и методы.

```
enum class ColorRGB( val r: Int, val g: Int, val b: Int ) {  
    RED(255, 0, 0),  
    RANGE(255, 165, 0),  
    YELLOW(255, 255, 0),  
    GREEN(0, 255, 0),  
    BLUE(0, 0, 255),  
    INDIGO(75, 0, 130),  
    VIOLET(238, 130, 238);  
  
    fun rgb() = (r * 256 + g) * 256 + b  
  
}  
  
fun main() {  
    println(ColorRGB.BLUE.rgb())  
}
```

Объявляя константу перечисления, необходимо указать значения её свойств. Обратите внимание, что на этом примере вы видите единственное место в синтаксисе *Kotlin*, где требуется использовать точку с запятой: когда в классе перечисления определяются какие-либо методы, точка с запятой отделяет список констант от определений методов.

Подобно *if*, оператор *when* - это выражение, возвращающее значение, поэтому можно написать функцию с телом-выражением, которая напрямую

возвращает выражение *when*.

```
fun getMnemonic (color: Color ) =
    when (color) {
        Color.RED -> "Каждый"
        Color.ORANGE -> "Охотник "
        Color.YELLOW -> "Желает"
        Color.GREEN -> "Знать"
        Color.BLUE -> "Где"
        Color.INDIGO -> "Сидит"
        Color.VIOLET -> "Фазан"
    }

println(getMnemonic(Color.BLUE))
```

В *Kotlin* не нужно добавлять в каждую ветку инструкцию *break*. При наличии совпадения выполнится только соответствующая ветка. В одну ветку можно объединить несколько значений, разделив их запятыми.

```
fun getNumber(color: Color) =
    when(color) {
        Color.RED, Color.ORANGE, Color.YELLOW -> 1
        Color.GREEN -> 0
        Color.BLUE, Color.INDIGO, Color.VIOLET -> -1
    }
```

Вы можете упростить код, импортировав значения констант.

```
package by.bstu.patsei.myapplication

import by.bstu.patsei.myapplication.Color.*

fun getNumber(color: Color) =
    when(color) {
        RED, ORANGE, YELLOW -> 1
        GREEN -> 0
        BLUE, INDIGO, VIOLET -> -1
    }
```

В отличие от *switch*, который требует использовать константы (константы перечисления, строки или числовые литералы) в определениях вариантов, оператор *when* позволяет использовать любые объекты.

```

fun mix(c1: Color, c2: Color) =
    when (setOf(c1, c2)) {
        setOf(RED, YELLOW) -> ORANGE
        setOf(YELLOW, BLUE) -> GREEN
        setOf(BLUE, VIOLET) -> INDIGO
        else -> throw Exception("не цвет")
    }

```

Стандартная библиотека *Kotlin* включает функцию *setOf*, которая создает множество *Set* с объектами, переданными в аргументах.

Выражение *when* последовательно сравнивает аргумент с условиями во всех ветвях, начиная с верхней, пока не обнаружит совпадение. Если ни одно из условий не выполнится, произойдет переход на ветку *else*.

### *Приведение типа*

В *Kotlin* принадлежность переменной к определенному типу проверяется с помощью оператора *is*. Эта проверка подобна оператору *instanceof* в *Java*. Но в *Java* для получения доступа к нужным свойствам и методам необходимо выполнить явное приведение после проверки оператором *instanceof*.

```

if (e is Sum) {
    return eval(e.right) + eval(e.left)
}

```

В *Kotlin* если вы проверили переменную на соответствие определенному типу, *приводить её к этому типу уже не надо; её можно использовать как значение проверенного типа*. Это автоматическое приведение типа (*smart cast*).

Автоматическое приведение работает, только если переменная не изменялась после проверки оператором *is*. Автоматическое приведение применяется *только к свойствам класса, объявленным с ключевым словом val и не имеющим метода записи*. В противном случае нельзя гарантировать, что каждое обращение к объекту будет возвращать одинаковое значение.

Явное приведение к конкретному типу выражается с помощью ключевого слова *as*:

```

val n = e as Num

```

## 13.4 Итерации, циклы и интервалы

Цикл *while* действует так же, как в *Java*.

Цикл *for* существует только в одной форме, эквивалентной циклу *for-*

*each* в *Java*. Он записывается в форме *for* <элемент> *in* <элементы>, как в C#. Чаще всего этот цикл используется для обхода элементов коллекций - как в *Java*.

В языке *Kotlin* есть циклы *while* и *do-while*, и их синтаксис не отличается от синтаксиса соответствующих циклов в *Java*.

## Интервалы

Диапазон представляет собой интервал между двумя значениями, обычно числовыми: началом и концом. Диапазоны определяются с помощью оператора **..**

```
val oneToTen = 1..10
```

Диапазоны в *Kotlin* - **закрытые или включающие**, т. е. второе значение всегда является частью диапазона.

```
val healthStatus = when (healthPoints) {  
    10 -> "excellent"  
    in 7..9 -> "good"  
    in 5..6 -> "not bad"  
    in 1..4 -> "bad"  
    else -> "not specified" }
```

Оператор диапазона **..** работает не только для чисел, но и для символов.

```
for (c in 'A'..'F')
```

Но диапазоны не ограничиваются и символами. Если есть класс, который поддерживает сравнение экземпляров (за счет реализации интерфейса *java.lang.Comparable*), вы сможете создавать диапазоны из объектов этого типа, но не можете перечислить всех объектов в таких диапазонах.

```
println("Kotlin" in "Java".. "Scala") //true
```

Здесь строки сравниваются по алфавиту, потому что именно так класс *String* реализует интерфейс *Comparable*. Та же проверка *in* будет работать с коллекциями:

```
println("Kotlin" in setOf("Java", "Scala")) //false
```

Кроме оператора `..` существуют еще несколько функций создания интервалов. Функция ***downTo*** создает убывающий интервал.

```
for (i in 100 downTo 1 step 2)
```

Здесь выполняется обход прогрессии с шагом, что позволяет пропускать некоторые значения. Шаг также *может быть отрицательным* - в таком случае обход прогрессии будет выполняться в обратном направлении.

Функция ***until*** создает интервал, *не включающий верхнюю границу* выбранного диапазона.

```
for (x in 0 until size)
```

эквивалентно выражению

```
for (x in 0.. size-1)
```

, но выглядит проще.

Для проверки вхождения значения в диапазон можно использовать оператор ***in*** или его противоположность - ***!in***, проверяющий отсутствие значения в диапазоне.

```
fun isLetter(c: Char) = c in 'a'..'z' || c in 'A'..'Z'
```

```
fun isNotDigit(c: Char) = c !in '0'..'9'
```

## 13.5 Шаблонные строки

Строку можно сконструировать из значений переменных и даже из результатов условных выражений. Чтобы упростить эту задачу и сделать код более понятным, в *Kotlin* предусмотрены шаблонные строки. Шаблоны позволяют включать значения переменных в кавычки.

```
println(name + " " + age)
```

```
println("$name $age")
```

**\$** специальный символ сообщает *Kotlin*, что вы хотите включить ***val*** или ***var*** в определяемую строку. Обратите внимание, что шаблонные значения появляются внутри кавычек, определяющих строку.

*Kotlin* позволяет вычислить значение внутри строки и интерполировать результат, то есть внедрить его в строку. Значение любого выражения, заключенного в фигурные скобки после знака доллара (***\${}***), будет автоматически вставлено в строку.



```
println("(Color: $Color) " +
        "(Blessed: ${if (isBlessed) "YES" else "NO"})")
```

## 13.6 Строки

Неважно, *var* или *val*, все строки в языке *Kotlin* на самом деле **неизменяемые** (как и в *Java*).

Оператор структурного равенства `==` сравнивает строки посимвольно, то есть считает строки равными, если они содержат одни и те же символы, следующие в одном и том же порядке.

Равенство ссылок проверяется с помощью `===`.

```
"Kotlin".forEach {
    println("$it\n")
}
```

В *Kotlin* для преобразования строки в регулярное выражение используется функция-расширение *toRegex*.

```
val regex = """.(.+)/(.+)\. (.+) """.toRegex()
```

В этом примере регулярное выражение записано в тройных кавычках. В такой строке *не нужно экранировать символы*, включая обратную косую черту, поэтому литерал точки можно описать как `\.` вместо `\\.` в обычных строковых литералах.

Тройные кавычки нужны не только для того, чтобы избавиться от необходимости экранировать символы. Такой строковый литерал может содержать любые символы, включая переносы строк.

## 13.7 Исключения

Обработка исключений в *Kotlin* выполняется так же, как в *Java* и многих других языках. Функция может завершиться обычным способом или возбудить исключение в случае ошибки. Код, вызывающий функцию, может перехватить это исключение и обработать его; если этого не сделать, исключение продолжит свое движение вверх по стеку.

```
val percentage = if (number in 0..100) println("ok") else
throw IllegalArgumentException( "Must be between 0 and 100:
$number")
```

Как и в *Java*, для обработки исключений используется выражение *try* с разделами *catch* и *finally*.

Самое большое отличие от *Java* заключается в **отсутствии конструкции *throws*** в сигнатуре функции: в *Java* вам пришлось бы добавить *throws IOException* после объявления функции. Это необходимо, потому что исключение *IOException* является контролируемым. В *Java* такие исключения требуется обрабатывать явно.

*Kotlin* не делает различий между контролируемыми и неконтролируемыми исключениями. Исключения, возбуждаемые функцией, не указываются - можно обрабатывать или не обрабатывать любые исключения. Такое проектное решение основано на практике использования контролируемых исключений в *Java*. Как показал опыт, правила языка *Java* часто требуют писать массу бессмысленного кода для повторного возбуждения или игнорирования исключений, и эти правила не всегда в состоянии защитить вас от возможных ошибок.

Ключевое слово *try* в языке *Kotlin* наряду с *if* и *when* является выражением, значение которого можно присвоить переменной. В отличие от *if*, тело выражения всегда нужно заключать в фигурные скобки. Как и в остальных случаях, если тело содержит несколько выражений, итоговым результатом станет значение последнего выражения.

```
val number = try {
    Integer.parseInt(readLine().toString())
}
catch (e: NumberFormatException) {
    null
}
```

Если блок *try* выполнится нормально, последнее выражение в нём станет его результатом. Если возникнет исключение, результатом станет последнее выражение в соответствующем блоке *catch*.

## 13.8 Nullability

Во многих языках программирования, включая *Java*, *null* — это частая причина сбоев, потому что несуществующая величина не может выполнить работу. Некоторым элементам в *Kotlin* может быть присвоено значение *null*, а другим нет.

*Kotlin* принимает следующую позицию в обращении с *null*. **Если не указано иное, то переменной нельзя присвоить значение *null***. Чтобы разрешить любой тип, в том числе и *null*, мы должны указать это **явно**, добавив вопросительный знак после имени типа:

```
public fun readFromConsol(): String?{
return readLine();
}
```

Знак вопроса показывает, что тип поддерживает значение *null*. Это означает, что *readFromConsol* вернет значение *String* или *null*.

Никогда нельзя быть уверенным, что значение не вернет *null*. В таких случаях в первую очередь следует использовать **оператор безопасного вызова (?) функции**. Он сочетает в одной операции проверку на *null* и вызов метода.

```
var aver = readFromConsol()?.capitalize()
```

Когда компилятор встречает оператор безопасного вызова, он знает, что надо проверить значение на *null*. Обнаружив *null*, он пропустит вызов функции и просто вернет *null*. Например, если *aver* хранит значение, отличное от *null*. Если *aver* хранит *null*, функция *capitalize* не будет вызвана. В таких случаях, как в примере выше, мы говорим, что функция *capitalize* вызывается «безопасно» и риска *NullPointerException* не существует.

Если вы хотите выполнить дополнительную работу, например, ввести новое значение или вызвать другие функции, если значение переменной отличается от *null*? Один из способов достичь этого — использовать **оператор безопасного вызова с функцией *let***. *let* вызывается со значением, и суть в том, чтобы разрешить объявлять переменные для выбранной области видимости.

```
var bever = readFromConsol()?.let {
    if (it.isNotBlank()) {
        it.capitalize()
    } else {
        "No data"
    }
}

println(bever)
```

Когда *bever* имеет значение, отличное от *null*, вызывается *let* и выполняется тело анонимной функции, переданной в *let*: проверяются входные данные и определяется, было ли введено что-то пользователем; если да, то первая буква становится прописной, а если нет, то возвращается *"No data"*.

Функции *isNotBlank* и *capitalize* требуют, чтобы аргумент имел тип, не поддерживающий *null*, что обеспечит функция *let*.

Чтобы вызвать функцию для переменной, тип которой поддерживает значение *null*, также можно использовать оператор **!!**.

Визуально **!!** должен выделяться в коде, потому что это *опасный вариант*. Он **преобразует любое значение к типу, не поддерживающему значения *null***. Использовать **!!** — все равно что сказать компилятору: «Если я хочу провести операцию с несуществующим значением, то ТРЕБУЮ, чтобы ты вызвал *NullPointerException*!» (кстати говоря, этот оператор также называется **оператором контроля non-null**, но чаще просто оператором двойного восклицательного знака).

```
aver = readFromConsol()!!.capitalize()
```

читается так: «Мне все равно, что *aver* может быть *null*. Вызвать *capitalize*!». Однако если *он* будет иметь значение *null*, вы получите *NullPointerException*.

Если вы абсолютно точно уверены, что переменная не станет *null*, то применение оператора **!!** может быть неплохим выходом.

Третий вариант для безопасной работы с *null*-значениями — это проверить переменную с помощью оператора *if*.

Оператор безопасного вызова предпочтительнее *value != null* как более гибкий инструмент, решающий проблему меньшим количеством кода.

Еще один вариант проверки значения на *null* — это оператор **?:** (null coalescing operator, или **оператор объединения по null, также известный как оператор «Элвис»**). «Если операнд слева от меня — *null*, выполнить операцию справа».

```
val rage: String = readFromConsol() ?: "No data"
```

Оператор **?:** можно использовать в сочетании с функцией *let* вместо оператора *if/else*.

```
var beverage = readLine()
beverage?.let {
    beverage = it.capitalize()
} ?: println("Beverage was null!")
```

## 13.9 Система типов Kotlin

### Встроенные типы языка Kotlin

Все числовые типы в *Kotlin*, как и в *Java*, имеют знак, то есть они могут представлять *положительные и отрицательные числа*.

В табл. перечислены наиболее часто используемые типы, доступные в *Kotlin*.

Тип	Битов	Макс. значение	Мин. значение
Byte	8	127	-128
Short	16	32767	-32768
Int	32	2147483647	-2147483648
Long	64	9223372036854775807	-9223372036854775808
Float	32	3.4028235E38	1.4E-45
Double	64	1.7976931348623157E308	4.9E-324

*Kotlin* не различает примитивных типов и типов-обертков. Вы всегда используете один и тот же тип

В большинстве случаев - для переменных, свойств, параметров и возвращаемых значений - тип *Int* в *Kotlin* компилируется в примитивный тип *int* из *Java*. Единственный случай, когда это невозможно, - обобщенные классы, например коллекции. Примитивный тип, указанный в качестве аргумента типа обобщенного класса, компилируется в соответствующий тип-обертку в *Java*. Поэтому если в качестве аргумента типа коллекции указан *Int*, то коллекция будет хранить экземпляры соответствующего типа-обертки *java.lang.Integer*.

Такие *Kotlin*-типы, как *Int*, за кулисами могут компилироваться в соответствующие примитивные типы *Java*, поскольку значения обоих типов не могут хранить ссылку на *null*. Обратное преобразование работает аналогично: при использовании *Java*-объявлений в *Kotlin* примитивные типы становятся типами, не допускающими *null* (а не платформенными типами), поскольку они не могут содержать значения *null*.

*Kotlin*-типы с поддержкой *null* не могут быть представлены в *Java* как примитивные типы, поскольку в *Java* значение *null* может храниться только в переменных ссылочных типов. Это означает, что всякий раз, когда в коде на *Kotlin* используется версия простого типа, допускающая значение *null*, она компилируется в соответствующий тип-обертку.

Чаще других типов используются следующие:

Тип	Описание	Пример
String (строка)	Текстовая информация	"Estragon" "happy meal"
Char (символ)	Один символ	'x' Символ Юникод U+0041
Boolean (логический)	Истинно/ложно Да/Нет	true false
Int (целочисленный)	Целое число	"Estragon".length 5
Double (с плавающей запятой)	Дробные числа	3.14 2.718
List (список)	Коллекция элементов	3, 1, 2, 4, 3 "root beer", "mclub soda", "coke"
Set (множество)	Коллекция уникальных значений	"Larry", "Moe", "Curly" "Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune"
Map (ассоциативный массив)	Коллекция пар «ключ-значение»	"small" to 5.99, "medium" to 7.99, "large" to 10.99

## Преобразование типов в Kotlin

Одно важное отличие *Kotlin* от *Java* - способ обработки числовых преобразований. *Kotlin не выполняет автоматического преобразования чисел из одного типа в другой*, даже когда другой тип охватывает более широкий диапазон значений. Например, в *Kotlin* следующий код не будет компилироваться:

```
val i = 1
val I: Long = i // error
```

Вместо этого нужно применить явное преобразование:

```
val i = 1
val I: Long = i.toLong()
```

Функции преобразования определены для каждого простого типа (кроме *Boolean*): *toByte()*, *toShort()*, *toChar()* и т. д. Функции поддерживают преобразование в обоих направлениях: расширение меньшего типа к большему, как *Int.toLong()*, и усечение большего типа до меньшего, как *Long.toInt()*. *Kotlin* делает преобразование явным, чтобы избежать неприятных неожиданностей, особенно при сравнении обернутых значений.

## Корневые типы Any и Any?

Подобно тому, как тип *Object* является корнем иерархии классов в *Java*, тип *Any* - это супертип всех типов в *Kotlin*, не поддерживающих *null*. Но в *Java* тип *Object* - это супертип для всех ссылочных типов, а примитивные типы не являются частью его иерархии. Это означает, что когда требуется экземпляр *Object*, то для представления значений примитивных типов нужно использовать типы-обертки, такие как *java.lang.Integer*. В *Kotlin* тип *Any* - супертип для всех типов, в том числе для примитивных, таких как *Int*.

Так же как в *Java*, присваивание примитивного значения переменной типа *Any* вызывает автоматическую упаковку значения:

```
val answer: Any = 42
```

тип *Any* не поддерживает значения *null*.

Если нужна переменная, способная хранить любое допустимое значение в *Kotlin*, в том числе *null*, используйте тип *Any?*. На уровне реализации тип *Any* соответствует типу *java.lang.Object*.

## Тип Unit: тип «отсутствующего» значения

Тип *Unit* играет в *Kotlin* ту же роль, что и *void* в *Java*. Он может использоваться в качестве типа возвращаемого значения функции, которая не возвращает ничего.

```
fun f() { ... }
```

Если ваша *Kotlin*-функция объявлена как возвращающая тип *Unit* и она не переопределяет обобщенную функцию, она будет скомпилирована в старую добрую функцию *void*. Если вы переопределяете её в *Java*, то переопределяющая функция просто должна возвращать *void*.

В отличие от *void*, тип *Unit* - это полноценный тип, который может использоваться как аргумент типа.

Имя *Unit* традиционно используется в функциональных языках и означает «единственный экземпляр», а это именно то, что отличает тип *Unit* в *Kotlin* от *void* в *Java*.

## Тип Nothing

Для некоторых функций в *Kotlin* понятие возвращаемого значения просто не имеет смысла, поскольку они никогда не возвращают управления.



При анализе кода, вызывающего такую функцию, полезно знать, что она *не возвращает управления*. Чтобы выразить это, в Kotlin используется специальный тип возвращаемого значения *Nothing*:

```
fun fail(message: String): Nothing {  
    throw IllegalStateException(message)  
}
```

Тип *Nothing* не имеет значений, поэтому его имеет смысл использовать только в качестве типа возвращаемого значения функции или аргумента типа для обозначения типа возвращаемого значения обобщенной функции.

## 13.10

## Функции

Первая часть функции — это заголовок. Заголовок функции состоит из пяти частей: модификатора видимости, ключевого слова объявления функции, имени функции, параметров функции, типа возвращаемого значения

```
private fun formatHealthStatus  
(healthPoints: Int = 8, isBlessed: Boolean = true): String {  
    val healthStatus = when (healthPoints) {  
        ...  
    }  
    return healthStatus  
}
```

*Модификатор видимости*

**По умолчанию функция получает глобальную видимость (*public*)** — это означает, что все остальные функции (даже функции, объявленные в других файлах проекта) могут использовать эту функцию. Другими словами, если вы не указали модификатор, будет считаться, что используется модификатор «public».

*Параметры функции*

Функции могут требовать от нуля до нескольких и более параметров. Обратите внимание, что параметры функции всегда доступны только для чтения, то есть в теле функции они не могут менять свои значения. Другими словами, в теле функции параметры — *это val, а не var*.

*Тип возвращаемого значения*

Последняя часть заголовка функции — это тип возвращаемого значения, который определяет тип выходных данных функции после завершения



ее работы.

Локальная переменная *healthStatus* существует только в теле функции *formatHealthStatus*, то есть существует только в области видимости функции

```
private fun formatHealthStatus(healthPoints: Int, isBlessed:
Boolean): String {
    val healthStatus = when (healthPoints) {
        ...
    }
    return healthStatus
}
```

Переменная уровня файла доступна из любого места в проекте (однако в объявление можно добавить модификатор видимости и изменить область видимости переменной).

```
const val MAX_TIME: Int = 50

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

Переменные уровня файла существуют, пока не завершится выполнение всей программы.

Из-за разницы между локальными переменными и переменными уровня файла компилятор выдвигает разные требования к тому, когда им должно присваиваться начальное значение. Значения переменным уровня файла должны присваиваться сразу при объявлении, иначе код не скомпилируется.

Так как локальная переменная имеет более ограниченную область применения - компилятор более снисходителен к тому, где она должна быть инициализирована, лишь бы она инициализировалась до ее использования.

Для функций с единственным выражением можно не указывать тип возвращаемого значения, фигурные скобки и оператор `return`.

```
fun castMark(mark: Int = 6) = println("Your mark is $mark")
```

### *Функции с возвращаемым типом Unit*

Не все функции возвращают значение. Некоторые производят иную работу, например, изменяют состояние переменной или вызывают другие функции, которые обеспечивают вывод данных.

В *Kotlin* подобные функции известны как функции с возвращаемым типом *Unit*. Для предыдущего примера

```

    mix();
    mix(YELLOW)
    mix(2 = RED)
    mix(c2 = GREEN, c1 = YELLOW)
    castMark()
}

fun castMark(mark: Int = 6) = println("Ypur mark is $mark")

```

Мышкой выберите функцию и нажмите (Control-Shift-P). Android Studio отобразит тип возвращаемого значения.

*Kotlin* использует тип *Unit*, чтобы обозначить функцию, которая не «возвращает» значения. Если ключевое слово *return* не используется, то считается, что функция возвращает значение типа *Unit*. Без типа невозможно было бы работать с обобщениями.

### 13.10.1 Именованные аргументы

Вызывая функции, написанные на *Kotlin*, можно указывать имена некоторых аргументов. Когда указано имя одного аргумента, то необходимо указать имена всех аргументов, следующих за ним, чтобы избежать путаницы.

```

joinToString(collection, separator = " ", prefix = " ", postfix = ".")

```

### 13.10.2 Аргументы по умолчанию

Другая распространенная проблема *Java* - избыток перегруженных методов в некоторых классах. Перегрузка может использоваться ради обратной совместимости, для удобства пользователей или по иным причинам, итог - дублирование.

В *Kotlin* часто можно избежать перегрузки благодаря возможности указывать значения параметров по умолчанию в объявлениях функций.

```

fun mix(c1: Color = RED, c2: Color = YELLOW) =
    when (setOf(c1, c2)) {
        setOf(RED, YELLOW) -> ORANGE
        setOf(YELLOW, BLUE) -> GREEN
        setOf(BLUE, VIOLET) -> INDIGO
        else -> throw Exception("не цвет")
    }

```

Теперь функцию можно вызвать с аргументами или опустить некоторые

из них:

```
mix();  
mix(YELLOW)
```

При использовании именованных аргументов можно опустить аргументы из середины списка и указать только нужные, причем в любом порядке:

```
mix(c2 = RED)  
mix(c2 = GREEN, c1 = YELLOW)
```

### 13.10.3 Свойства и функции верхнего уровня

*Java* как объектно-ориентированный язык требует помещать весь код в методах классов. В реальности почти во всех больших проектах есть много кода, который нельзя однозначно отнести к одному классу. В конечном итоге появляется множество классов, которые не имеют ни состояния, ни методов экземпляров и используются только как контейнеры для кучи статических методов.

В *Kotlin* не нужно создавать этих классов. Вместо этого можно помещать функции непосредственно на верхнем уровне файла, за пределами любых классов. Такие функции всё ещё остаются членами пакета, объявленного в начале файла, и их всё ещё нужно импортировать для использования в других пакетах, но ненужный дополнительный уровень вложенности исчезает.

Например файл *cast.kt* с содержимым

```
package by.bstu.patsei.myapplication  
fun castMark(mark: Int = 6) = "Your mark is $mark"
```

Поскольку JVM может выполнять только код в классах, то компилятор Kotlin генерирует имя класса, соответствующее имени файла с функцией. Все функции верхнего уровня в файле компилируются в статические методы этого класса.

```
public class CastKt{  
    public static String castMark (int mark){}  
}
```

Поэтому вызов такой функции из *Java* выглядит так же, как вызов любого другого статического метода:

```
CastKt.castMark(8);
```

Чтобы изменить имя класса с *Kotlin*-функциями верхнего уровня, нужно добавить в файл **аннотацию** `@JvmName`. Поместите её в начало файла перед именем пакета:

```
@file:JvmName("TempFunctions")
package by.bstu.patsei.myapplication
fun castMark(mark: Int = 6) = "Your mark is $mark"
```

Теперь вызов будет такой

```
TempFunctions.castMark(8);
```

Хранение отдельных фрагментов данных вне класса требуется не так часто, но все же бывает полезно.

```
var opCount = 0 //Объявление свойства верхнего уровня
fun performOperation() {
    opCount++
}
```

Значение такого свойства будет храниться в статическом поле.

### 13.10.4 Свойства и функции расширения

Одна из главных особенностей языка *Kotlin* - простота интеграции с существующим кодом. Даже проекты, написанные исключительно на языке *Kotlin*, строятся на основе *Java*-библиотек, таких как JDK, фреймворк Android и другие. И когда код на *Kotlin* интегрируется в *Java*-проект, то приходится иметь дело с существующим кодом, который не был или не будет переводиться на язык *Kotlin*. Для этого и созданы функции-расширения. По сути, **функция-расширение - это функция, которая может вызываться как член класса, но определена за его пределами.**

```
fun String.SecondChar(): Char =
    if (this.get(1).isDefined())
        this.get(1)
    else
        throw IndexOutOfBoundsException("out range")
println("Kotlin".SecondChar())
println("2".SecondChar())
```

Чтобы определить такую функцию, достаточно добавить имя расширяемого класса или интерфейса перед именем функции.

Функцию можно вызывать, используя тот же синтаксис, что и для обычных членов класса.

В этом примере класс *String* - это тип-получатель. Не важно даже, написан класс *String* на *java*, на *Kotlin* или другом JVM-языке (например, *Groovy*). Если он компилируется в *Java*-класс, мы можем добавлять в него свои расширения. В теле функции-расширения ключевое слово *this* используется так же, как в обычном методе.

Функции-расширения не позволяют нарушать правила инкапсуляции. В отличие от методов, объявленных в классе, функции-расширения *не имеют доступа к закрытым или защищенным членам класса*.

***Можно поменять имя импортируемого класса или функции, добавив ключевое слово as:***

```
import by.bstu.patsei.myapplication.SecondChar as second
fun main(args: Array<String>) {
    val c = "Kotlin".second()
}
```

*В Kotlin допускается переопределять функции-члены, но нельзя переопределить функцию-расширение.*

### *Свойства-расширения*

Хотя они называются свойствами, свойства-расширения не могут иметь состояние из-за отсутствия места для его хранения: нельзя добавить дополнительных полей в существующие экземпляры объектов *Java*. Поскольку отдельного поля для хранения значения не существует, метод чтения должен определяться всегда и не может иметь реализации по умолчанию. Инициализаторы не разрешены по той же причине: негде хранить указанное значение.

Определяя свойство для класса *StringBuilder*, его можно объявить как *var*, потому что содержимое экземпляра *StringBuilder* может меняться.

```
var StringBuilder.lastChar: Char
    get() = get(length - 1)
    set(value: Char) {
        this.setCharAt(length - 1, value)
    }
```

Функции-расширения и свойства-расширения дают возможность расширять API любых классов, в том числе классов во внешних библиотеках, без модификации их исходного кода и без дополнительных накладных расходов во время выполнения.

### 13.10.5 Функции, принимающие произвольное число аргументов

*Kotlin* использует модификатор *vararg* перед параметром функции для передачи переменного числа.

```
fun sumOf (vararg values: Int): Int{
    var sum = 0
    for (n in values)
        sum+=n
    return sum;
}

println(sumOf(3, 6, 7, 123, 3, 4)) //146
```

### 13.10.6 Локальные функции

Функции, извлеченные из основной функции, можно сделать вложенными

```
class User(val id: Int, val name: String, val address: String)

fun saveUser(user: User) {

    fun validate(user: User, value: String, fieldName:
String) {
        if (value.isEmpty()) {
            throw
IllegalArgumentException( "Can't save user ${user.id}: empty
$fieldName")
        }
    }
    validate(user, user.name, "Name")
    validate(user, user.address, "Address")
    // Сохранение информации о пользователе в базе данных
}
```

Этим можно избежать дублирования и сохранить четкую структуру кода.

Локальные функции имеют доступ ко всем параметрам и переменным охватывающей функции. Поэтому *User* можно убрать из параметров

```

class User(val id: Int, val name: String, val address: String)

fun saveUser(user: User) {

    fun validate( value: String, fieldName: String) {
        if (value.isEmpty()) {
            throw IllegalArgumentException( "Can't save
user ${user.id}: empty $fieldName")
        }
    }
    validate( user.name, "Name")
    validate( user.address, "Address")
    // Сохранение информации о пользователе в базе данных
}

```

Можно ещё улучшить этот пример, если перенести логику проверки в функцию-расширение класса User.

```

class User(val id: Int, val name: String, val address: String)

fun User.validateBeforeSave() {

    fun validate(value: String, fieldName: String) {
        if (value.isEmpty()) {
            throw IllegalArgumentException( "Can't save user
$id: empty $fieldName")
        }
        validate(name, "Name")
        validate(address, "Address")
    }

    fun saveUser(user: User) {
        user.validateBeforeSave()
    }

}

```

### 13.10.7 Функции высшего порядка

Функциями высшего порядка называют *функции, которые принимают другие функции в аргументах и/или возвращают их*. В Kotlin функции могут быть представлены как обычные значения, в виде лямбда-выражений или ссылок на функции. То есть функция высшего порядка - это любая функция, которая принимает аргумент с лямбда-выражением или ссылкой на функцию и/или возвращает их.

Рассмотрим функцию

```
fun main(args: Array<String>) {
    val greetingFunction: () -> String = {
        "Welcome to Kotlin"
    }

    println(greetingFunction())
}
```

Это функциональный *fun : () -> String*, который сообщает компилятору, какой тип функции может содержаться в переменной.

Объявление функционального типа состоит из двух частей: параметров функции в скобках и возвращаемого типа, следующего за стрелкой (->).

Объявление типа переменной *greetingFunction: () -> String* показывает, что компилятор может присвоить *greetingFunction* любую функцию, которая не принимает аргументов (пустые скобки) и возвращает *String*.

В отличие от именованной функции, анонимная функция не требует — а в редких случаях даже запрещает — использовать ключевое слово *return* для возврата данных. Анонимные функции неявно, или автоматически, возвращают результат выполнения последней инструкции в теле функции, позволяя отбросить ключевое слово *return*.

```
val greetingFunction: () -> String = {
—return "Welcome to Kotlin"
}
```

Эта особенность анонимных функций и удобна, и нужна для их синтаксиса. Ключевое слово *return* запрещено в анонимной функции, так как это создает двусмысленность для компилятора: из какой функции вернуть значение — из функции, в которой была вызвана анонимная функция, или из самой анонимной функции.

### Ключевое слово *it*

В анонимной функции, которая принимает ровно один аргумент, вместо определения имени параметра можно использовать удобную альтернативу — ключевое слово *it*. В анонимных функциях с одним параметром можно использовать и именованный параметр, и его замену — ключевое слово *it*.

```
fun main(args: Array<String>) {
    val greetingFunction: (String) -> String = {
        "$it, Welcome to Kotlin"
    }
}
```



```

    }

    println(greetingFunction("FIT")) // FIT, Welcome
to Kotlin
}

```

Ключевое слово *it* можно использовать в анонимной функции, принимающей один аргумент, но нельзя, если число аргументов больше одного. Рассмотрим еще пример

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

Обратите внимание, что в теле лямбда-выражения { x, y->x + y } можно опустить типы параметров x и y. Поскольку они указаны в объявлении типа функции, их не нужно повторять в самом лямбда-выражении. Точно как в любой другой функции, тип возвращаемого значения в объявлении типа функции можно отметить как допускающий значение *null*:

```
var canRetNull: (Int, Int) -> Int? = { x, y-> null }
```

Также можно определить переменную, которая может принимать значение *null* и относиться к типу функции. Чтобы указать, что именно переменная, а не функция способна принимать значение *null*, нужно заключить определение типа функции в круглые скобки и добавить знак вопроса в конце:

```
var funOrNull: ((Int, Int) -> Int)? = null
```

### 13.10.8 Ссылка на функцию

Лямбды можно передавать в аргументе другой функции.

```
inline fun runSimulation(playerName: String,
                          costPrinter: (Int) -> Unit,
                          greetingFunction: (String, Int) ->
String) {

    val numBuildings = (1..3).shuffled().last()
                          // Случайно выберет 1, 2 или 3
    costPrinter(numBuildings)
    println(greetingFunction(playerName, numBuildings))
}

```

Сделать можно иначе: передать ссылку на функцию. Ссылка на функцию преобразует именованную функцию (функцию, объявленную с ключевым словом `fun`) в значение, которое можно передавать как аргумент. Ссылку на функцию можно использовать везде, где допускается лямбда-выражение.

Чтобы получить ссылку на функцию, используйте оператор `::` с именем этой функции.

### 13.10.9 Тип функции как возвращаемый тип

Как и любой другой тип, функциональный тип также может быть возвращаемым типом. Это значит, что можно объявить функцию, которая возвращает функцию.

```
fun configureGreetingFunction(): (String) -> String {  
  
    val structureType = "hospitals"  
    var numBuildings = 5  
  
    return { playerName: String ->  
        val currentYear = 2022  
        numBuildings += 1  
        println("Adding $numBuildings $structureType")  
        "Welcome to Minsk, $playerName! (copyright $currentYear) "  
    }  
}  
  
val result = configureGreetingFunction()("ddf")  
println(configureGreetingFunction()("afsfds"))  
println(result)
```

*configureGreetingFunction* — это такая «фабрика функций», которая настраивает другую функцию. Она объявляет необходимые переменные и собирает их в лямбду, которую затем возвращает в точку вызов.

*numBuildings* и *structureType* — локальные переменные, объявленные внутри *configureGreetingFunction*, и обе используются в лямбде, возвращаемой из *configureGreetingFunction*, хотя объявляются во внешней функции, которая возвращает лямбду. Это возможно, потому что лямбды в Kotlin — это замыкания. Они замыкают переменные из внешней области видимости, в которой были определены.

Функция, которая принимает или возвращает другую функцию, также называется функцией высшего порядка. Терминология позаимствована из

той же математической области, что и лямбда. Функции высшего порядка используются в функциональном программировании.

### 13.10.10 Стандартные функции

Стандартные функции — это универсальные вспомогательные функции из стандартной библиотеки *Kotlin*, которые принимают лямбда-выражения, уточняющие их поведение.

Стандартные функции *Kotlin* на самом деле — функции-расширения, контекстом для которых служит объект-получатель.

*apply*

***apply* можно считать функцией настройки:** она позволяет вызывать несколько функций для объекта-получателя и настроить его для дальнейшего использования.

```
val menuFile = File("menu-file.txt")
menuFile.setReadable(true)
menuFile.setWritable(true)
menuFile.setExecutable(false)
```

Используя *apply*, то же самое можно реализовать меньшим количеством кода:

```
val menuFile = File("menu-file.txt").apply {
    setReadable(true)
    setWritable(true)
    setExecutable(false)
}
```

*apply* позволяет отбросить имя переменной в каждом вызове функции, выполняемом для настройки объекта-получателя, потому что все функции в лямбде вызываются относительно объекта-приемника, для которого вызвана сама функция.

*let*

*let* определяет переменную в области видимости заданной лямбды и позволяет использовать ключевое слово *it*

```
val firstItemSquared = listOf(1, 2, 3).first().let {
    it * it
}
```

возводит в квадрат первое число в списке

*run*

Функция *run* похожа на *apply*, точно так же ограничивая относительную область видимости, но не возвращает объект-приемник.

Например, вот как можно проверить наличие конкретной строки в файле:

```
val menuFile = File("menu-file.txt")
val contain = menuFile.run {
    readText().contains("Kotlin")
}
```

Функция *readText* неявно вызывается относительно объекта-приемника — экземпляра *File* — подобно функциям *setReadable*, *setWritable* и *setExecutable* в примере с *apply*. Но в отличие от *apply*, *run* возвращает результат лямбды — в нашем случае истину или ложь.

*run* может также использоваться для выполнения ссылки на функцию относительно объекта-приемника

```
fun nameIsLong(name: String) = name.length >= 20
"Kotlin".run(::nameIsLong) // ЛОЖЬ
```

Есть другая форма вызова *run*, без объекта-приемника. Эта форма встречается гораздо реже:

```
val healthPoints = 90;
val status = run {
    if (healthPoints == 100) "perfect health" else "has injuries"
}
```

*with*

*with* — это разновидность *run*. Она ведет себя похожим образом, но использует другие соглашения вызова. В отличие от стандартных функций, рассмотренных ранее, *with* требует, чтобы объект-приемник передавался ей в первом аргументе, а не как субъект вызова

```
val nameTooLong = with("Hello, World") {
    length >= 20
}
```

Такая несогласованность с остальными стандартными функциями делает *with* менее предпочтительной, чем *run*.

### *also*

Функция *also* похожа на функцию *let*. Как и *let*, *also* передает объект-приемник как аргумент в лямбду. Но есть одно большое различие между *let* и *also*: вторая возвращает объект-приемник, а не результат лямбды.

Пример ниже дважды вызывает *also* для выполнения двух разных операций: первая выводит имя файла, а вторая записывает содержимое файла

```
var fileContents: List<String>
File("file.txt")
    .also {
        print(it.name)
    }.also {
        fileContents = it.readLines()
    }
```

### *takeIf*

*takeIf* работает немного иначе, чем другие стандартные функции: она вычисляет условие, или предикат, заданное в лямбде, которое возвращает истинное или ложное значение. Если условие истинно, *takeIf* вернет объект-приемник. Если условие ложно, она вернет *null*.

Рассмотрите следующий пример, который читает файл, только если файл доступен для чтения и для записи:

```
val fileContents = File("myfile.txt")
    .takeIf { it.canRead() && it.canWrite() }
    ?.readText()
```

*takeIf* удобно использовать для проверки условия перед присваиванием значения переменной или продолжением работы. Концептуально *takeIf* — это оператор *if*, но с преимуществом прямого воздействия на экземпляр, что часто позволяет избавиться от временной переменной.

### *takeUnless*

Функция *takeUnless* действует так же, как *takeIf*, но возвращает объект-приемник, если условие ложно. Следующий код читает файл, если он не скрытый (и возвращает *null*, если скрытый):

```
val fileContents = File("myfile.txt").takeUnless {
    it.isHidden
}?..readText()
```

Функция	Передаёт объект-приемник в лямбду как аргумент?	Ограничивает относительную область видимости?	Возвращает
let	Да	Нет	Результат лямбды
apply	Нет	Да	Объект-приемник
run <sup>a</sup>	Нет	Да	Результат лямбды
with <sup>b</sup>	Нет	Да	Результат лямбды
also	Да	Нет	Объект-приемник
takeIf	Да	Нет	Версию объекта-приемника с поддержкой null
takeUnless	Да	Нет	Версию объекта-приемника с поддержкой null

### 13.10.12 Лямбда выражения

Лямбда-выражение представляет небольшой фрагмент поведения, которое можно передать как значение. Его можно объявить отдельно и сохранить в переменной. Но чаще оно объявляется непосредственно при передаче в функцию.

Лямбда-выражения в Kotlin всегда окружены фигурными скобками. Обратите внимание на отсутствие круглых скобок вокруг аргументов. Список аргументов отделяется от тела лямбда-выражения стрелкой.

```
val sum = { x: Int, y: Int -> x + y }
println(sum(1, 2))
```

Лямбда-выражение можно вызывать напрямую:

```
{ println(42) }()
```

Если нужно заключить фрагмент кода в блок, используйте библиотечную функцию `run`, которая выполнит переданное ей лямбда-выражение

```
run{ println(42) }
```

### 13.10.13 Встраиваемые функции

Лямбда-выражения обычно компилируются в анонимные классы. Но это означает, что каждый раз, когда используется лямбда-выражение, создается дополнительный класс; и если лямбда-выражение хранит какие-то переменные, для каждого вызова создается новый объект. Это влечет дополнительные накладные расходы, ухудшающие эффективность реализации с лямбда-выражениями по сравнению с функцией, которая выполняет тот же код непосредственно.

Если отметить функцию модификатором `inline`, компилятор не будет генерировать вызов функции в месте её использования, а просто вставит код её реализации.

Встраиваемые функции дают возможность производить нелокальный возврат, когда выражение `return` в лямбда-выражении производит выход из вмещающей функции.

## 13.11 Коллекции

Kotlin использует стандартные классы коллекций из Java. Все, что вы знаете о коллекциях в Java, верно и тут.

### 13.11.1 Изменяемые и неизменяемые коллекции

Важная черта, отличающая коллекции в Kotlin от коллекций в Java, - разделение интерфейсов, открывающих доступ к данным в коллекции только для чтения и для изменения. Это разделение начинается с базового интерфейса коллекций - *kotlin.collections.Collection*.

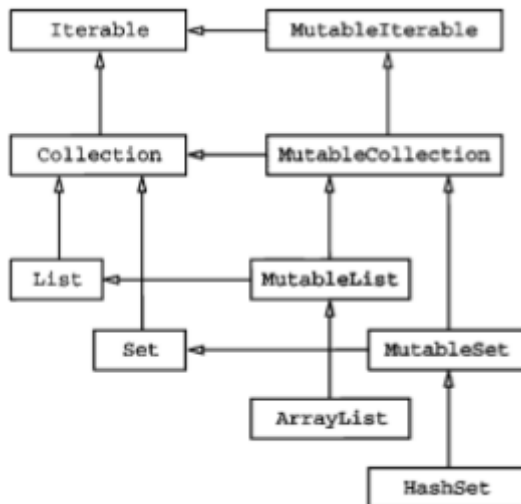
Чтобы получить возможность изменения данных в коллекции, используйте интерфейс *kotlin.collections.MutableCollection*. Он наследует интерфейс *kotlin.collections.Collection*, добавляя к нему методы для добавления и удаления элементов, очистки коллекции и т. д.

Как правило, вы везде должны использовать интерфейсы с доступом только для чтения. Применяйте изменяемые варианты, только если собираетесь изменять коллекцию.

```
val source: Collection<Int> = arrayListOf(3, 5, 7)
```

```
val target: MutableCollection<Int> = arrayListOf(1)
```

Любая коллекция в *Kotlin* - экземпляр соответствующего интерфейса коллекции Java. При переходе между *Kotlin* и *Java* никакого преобразования не происходит, и нет необходимости ни в создании оберток, ни в копировании данных. Но для каждого интерфейса *Java*-коллекций в *Kotlin* существуют два представления: только для чтения и для чтения/ записи



В *kotlin* часто используют три коллекции:

Тип коллекции	Только для чтения	Изменяемая коллекция
List	listOf	mutableListOf, arrayListOf
Set	setOf	mutableSetOf, hashSetOf, linkedSetOf, sortedSetOf
Map	mapOf	mutableMapOf, hashMapOf, linkedMapOf, sortedMapOf

Хотя коллекции в *Kotlin* представлены теми же классами, что и в *Java*, в *Kotlin* они обладают гораздо более широкими возможностями.

```

val strings = listOf("1", "2", "3") // создание списка
println(strings.last()) //3

val numbers = setOf(1, 14, 2) // множество
println(numbers.max()?.plus(numbers.sum())) //14+17 = 31

val map = hashMapOf(1 to "one", 7 to "seven", 53 to "fifty-three")
println(map.getValue(7)) // seven
  
```

В *Java*-коллекциях есть реализация по умолчанию для метода *toString*, но её формат вывода фиксирован и не всегда подходит



```
val strings = listOf("1", "2", "3") // создание списка
println(strings) //[1, 2, 3]
```

Рассмотрим создание словаря

```
val map = mapOf(1 to "one", 7 to "seven", 53 to "fifty-
three")
```

Ключевое слово `to` в этой строке - не встроенная конструкция, а специальная форма вызова метода, называемая инфиксным вызовом. В инфиксном вызове имя метода помещается между именем целевого объекта и параметром, без дополнительных разделителей.

Можно создавать параметризированные коллекции

```
val patronList: List<String> = listOf("Eli", "Mordoc",
"Sophie")
```

Любой элемент списка можно получить по его индексу, с помощью оператора `[]`.

*List* также обеспечивает другие удобные функции доступа по индексу, например, для извлечения первого и последнего элемента:

```
patronList.first() // Eli
patronList.last() // Sophie
```

Так как доступ к элементу по индексу может привести к появлению исключения, *Kotlin* обеспечивает функции безопасного доступа по индексу. Если индекс не существует, вместо исключения они могут вернуть какой-нибудь другой результат.

```
val patronList = listOf("Eli", "Mordoc", "Sophie")
patronList.getOrElse(4) { "Unknown Patron" }
```

Другая функция безопасного доступа, *getOrNull*, возвращает *null* вместо исключения.

Используйте функцию *contains* для проверки наличия элемента

```
if (patronList.contains("Eli")) ...
if (patronList.containsAll(listOf("Sophie", "Mordoc"))) ...
```

В языке *Kotlin* модифицируемый список известен как изменяемый список, и вы должны вызвать функцию *mutableListOf*, чтобы его создать.

```
val patronList = mutableListOf("Eli", "Mordoc", "Sophie")
println(patronList)
patronList.remove("Eli")
patronList.add("Alex")
println(patronList)
```

```
[Eli, Mordoc, Sophie]
[Mordoc, Sophie, Alex]
```

Еще про добавление и удаление

```
patronList.add(0, "Alex")

mutableListOf("Eli", "Mordoc", "Sophie") += "Reginald"
//[Eli, Mordoc, Sophie, Reginald]

mutableListOf("Eli", "Mordoc", "Sophie") += listOf("Alex",
"Shruti")
//[Eli, Mordoc, Sophie, Alex, Shruti]

val patronList = mutableListOf("Eli", "Mordoc", "Sophie")
patronList.removeIf { it.contains("o") }
//[Eli]
```

List имеет функции, позволяющие превращать неизменяемые списки в изменяемые и обратно прямо в процессе выполнения: *toList* и *toMutableList*.

```
val patronList = mutableListOf("Eli", "Mordoc", "Sophie")
val readOnlyPatronList = patronList.toList()
```

Очистка

```
mutableListOf("Eli", "Mordoc", "Sophie").clear()
```

В языке *Kotlin* все циклы *for* опираются на итерации. В языках Java и C# им эквивалентны циклы *foreach*.

```
for (patron in patronList) { ... }
```

Цикл *for* прост и легко читается, но если вы предпочитаете более функциональный стиль, используйте функцию *forEach*.

Функция *forEach* обходит каждый элемент в списке — последовательно от начала до конца — и передает каждый элемент анонимной функции в аргументе.

```
patronList.forEach { patron ->
    println("Good evening, $patron")
}
```

Цикл *for* и функция *forEach* функционально эквивалентны. Цикл *for* и функция *forEach* обрабатывают индексы неявно. Если вам потребуется получить индекс каждого элемента в списке, используйте *forEachIndexed*.

```
patronList.forEachIndexed { index, patron ->
    println("Good evening, $patron - you're #${index + 1}
in line.")
}
```

Функции *forEach* и *forEachIndexed* доступны и для некоторых других типов в *Kotlin*. Эта категория типов называется *Iterable* (итерируемые) и включает *List*, *Set*, *Map*, *IntRange*.

### 13.11.2 Деструктуризация

Список также предлагает возможность деструктуризации до пяти первых элементов. Позволяет объявить несколько переменных и присвоить им значения в одном выражении.

```
val menuData = "soup,harcho,45"
fun main() {
    val (type, name, price) = menuData.split()
    println("$type , $name, $price") //soup , harcho, 45
}
```

Можно выборочно деструктурировать элементы из списка, используя символ `_` для пропуска нежелательных элементов.

```
val menuData = listOf("soup", "harcho", "45")
fun main() {
    val (type, _, price) = menuData
    println("$type , $price") //soup , 45
}
```

### 13.11.3. Set

Множества удобны для генерации уникальных значений. Например, надо сгенерировать имена, таким образом, чтобы они не пересекались

```
val lastName = listOf("Ivanoff", "Petrov", "Shaz")
val firstName = listOf("Ivan", "Petia", "Nastia")
val uniquePatrons = mutableSetOf<String>()

fun main(args: Array<String>) {
    (0..9).forEach {
        val first = firstName.shuffled().first()
        val last = lastName.shuffled().first()
        val name = "$first $last"
        println(name)
        uniquePatrons += name
    }
    println(uniquePatrons)
}
```

## Результат

```
Ivan Shaz
Petia Petrov
Ivan Ivanoff
Petia Shaz
Ivan Ivanoff
Nastia Shaz
Petia Shaz
Nastia Petrov
Nastia Petrov
Nastia Petrov
[Ivan Shaz, Petia Petrov, Ivan Ivanoff, Petia Shaz, Nastia Shaz, Nastia Petrov]
```

Обратите внимание, что вы не можете рассчитывать на автоматическое определение типов для `uniquePatrons`, потому что объявили его как пустое множество. Необходимо указывать тип элементов, которое оно может содержать: `mutableSetOf<String>`.

В множестве содержатся только уникальные значения, поэтому вы наверняка получите менее 10 имен.

### 13.11.4 Преобразование коллекций

Преобразовать список в множество и обратно можно с помощью функций `toSet` и `toList` (или их изменяемых вариантов: `toMutableSet` и

*toMutableList*). Распространенный трюк — вызвать *toSet*, чтобы отбросить неуникальные элементы в списке.

```
listOf("Nastia Petrov", "Nastia Petrov", "Nastia Petrov").toSet().toList() // [Nastia Petrov]
```

или так

```
val patrons = listOf("Nastia Petrov", "Nastia Petrov", "Nastia Petrov").distinct() // // [Nastia Petrov]
```

### 13.11.5 Типы примитивных массивов

Как и с другими типами, аргумент типа в типе массива всегда становится ссылочным типом. Поэтому объявление *Array<Int>*, например, превратится в массив оберток для целых чисел (Java-тип `java.lang.Integer []`). Если вам нужно создать массив значений примитивного типа без оберток, используйте один из специализированных классов для представления массивов примитивных типов. Для этой цели в Kotlin есть ряд отдельных классов, по одному для каждого примитивного типа.

*Kotlin* включает несколько ссылочных типов с именами, включающими слово *Array*, которые компилируются в элементарные массивы Java. Типы *Array* используются главным образом для взаимодействий между кодом на *Java* и *Kotlin*.

```
val playerAges: IntArray = intArrayOf(34, 27, 14, 52, 101)
```

Тип *IntArray* представляет последовательность элементов, а именно целых чисел. В отличие от *List*, *IntArray* преобразуется в элементарный массив при компиляции. После компиляции получившийся байт-код будет точно совпадать с ожидаемым примитивным массивом `int`.

Преобразовать коллекцию *Kotlin* в соответствующий элементарный *Java* массив можно с помощью встроенных функций. Список целых чисел можно преобразовать в *IntArray*, используя функцию *toIntArray*, поддерживаемую типом *List*. Это позволит вам преобразовать коллекцию в массив `int`, когда понадобится передать элементарный массив в Java-функцию:

```
val playerAges: List<Int> = listOf(34, 27, 14, 52, 101)
playerAges.toIntArray()
```

Тип массива	Создающая функция
IntArray	intArrayOf
DoubleArray	doubleArrayOf
LongArray	longArrayOf
ShortArray	shortArrayOf
ByteArray	byteArrayOf
FloatArray	floatArrayOf
BooleanArray	booleanArrayOf
Array*	arrayOf

Кроме основных операций (получения длины массива, чтения и изменения элементов), стандартная библиотека Kotlin поддерживает для массивов тот же набор функций-расширений, что и для коллекций.

### 13.11.6 Map

Ключи уникальны и определяют значения в ассоциативном массиве: значения, напротив, не обязательно должны быть уникальными. С этой точки зрения ассоциативные массивы обладают одним из свойств множеств: они гарантируют уникальность ключей, подобно элементам множеств. Ассоциативные массивы создаются с помощью функций: *mapOf* или *mutableMapOf*.

```
mapOf("Eli" to 10.75, "Mordoc" to 8.25, "Sophie" to 5.50)
```

Функция *to* преобразует операнды слева и справа в пару (Pair) — тип, представляющий группу из двух элементов.

Существует другой способ определения элементов

```
var patronGold = mapOf(Pair("Eli", 10.75),
    Pair("Mordoc", 8.00),
    Pair("Sophie", 5.50))
```

добавить можно используя *+=*, но только уникальные ключи

```
patronGold += "Sophie" to 6.0
```

Так как ключ уже есть в массиве, существующая пара была затерта новой.

Получить доступ к значению в массиве можно по его ключу.

```
println(patronGold["Sophie"]) // 6.0
```

Для доступа можно использовать функции *getValue*, *getOrElse*, *getOrElseDefault*

Итого:

Тип коллекции	Упорядоченная?	Уникальные значения?	Хранит	Поддерживает деструктуризацию?
Список (List)	Да	Нет	Элементы	Да
Множество (Set)	Нет	Да	Элементы	Нет
Ассоциативный массив (Map)	Нет	Ключи	Пары «ключ-значение»	Нет

### 13.11.7 Функциональный API для работы с коллекциями

Функция *filter* выполняет обход коллекции, отбирая только те элементы, для которых лямбда-выражение вернет *true*:

```
val list = listOf(1, 2, 3, 4)
println(list.filter { it % 2 == 0 }) // [2, 4]
```

Функция *filter* сможет удалить из коллекции ненужные элементы, но не сможет изменить их. Для преобразования элементов вам понадобится функция *map*. Функция *map* применяет заданную функцию к каждому элементу коллекции, объединяя результаты в новую коллекцию.

```
val list = listOf(1, 2, 3, 4)
println(list.map { it * it }) //[1, 4, 9, 16]
```

Функции *filterKeys* и *mapKeys* отбирают и преобразуют ключи словаря соответственно, тогда как *filterValues* и *mapValues* отбирают и преобразуют значения.

```
val numbers = mapOf(0 to "zero", 1 to "one")
println(numbers.mapValues { it.value.toUpperCase() })
```

*Применение предикатов к коллекциям*

Ещё одна распространенная задача - проверка всех элементов коллекции на соответствие определенному условию. В Kotlin эта задача решается с помощью функций *all* и *any*. Функция *count* проверяет, сколько элементов удовлетворяет предикату, а функция *find* возвращает первый подходящий элемент.

```
val list = listOf(1, 2, 3)
println(!list.all{ it == 3 }) //true
println(list.any{ it != 3 }) //true
```

Группировка значений в списке с функцией *groupBy*

```
val listabc = listOf("a", "ab", "b", "c", "cc")
println(listabc.groupBy(String::first)) //{a=[a], a=[ab],
b=[b], c=[c, cc]}
```

Обработка элементов вложенных коллекций: функции *flatMap* и *flatten*. Функция *flatMap* сначала преобразует (или отображает - тар) каждый элемент в коллекцию, согласно функции, переданной в аргументе, а затем собирает (или уплощает - *flattens*) несколько списков в один.

```
val strings = listOf("abc", "def")
println(strings.flatMap { it.toList() }) // [a, b, c, d, e, f]
```

*orNull*

В Kotlin 1.4 все изменилось: были добавлены две новые функции, *maxOrNull* и *minOrNull*, а старые функции *max* и *min* устарели.

```
// Kotlin 1.3
val listInt = listOf(20, 30, 33)
val mvalue : Int? = listInt.max()

//Kotlin 1.4
val maxvalue : Int? = list.maxOrNull()
```



Стандартная библиотека содержит два набора функций: обычные и соответствующие функции с суффиксом *orNull*. Например, *toInt* - это обычная функция, а *toIntOrNull* - новая функция.

Разница в том, что обычные групповые функции возвращают ненулевое значение или генерируют исключение, например, когда строка не может быть преобразована в целое число. Напротив, суффиксные функции *OrNull* возвращают *null*, если что-то пошло не так.

### *reduce* and *runningReduce*

*runningReduce* - это новая функция, добавленная в Kotlin 1.4. Существующий метод *reduce* вычисляет и анализирует каждый элемент, тогда как новый *runningReduce* не анализирует какие-либо элементы, пока вся операция не будет завершена.

Общая функция возвращает окончательный результат, тогда как *runningReduce* возвращает все промежуточные шаги.

```
(1..5).reduce(sum);
```