МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт Информационных технологий

Кафедра информационных технологий в атомной энергетике

ИТОГОВЫЙ ОТЧЕТ

по дисциплине «Разработка приложений на языке Котлин»

Студент группы ИКБО-50-23 Павлов Н. С.

_____

(подпись студента)

Принял старший преподаватель Золотухин С. А.

_____

(подпись руководителя)

Работа представлена «____»_____ 2025 г.

Допущен к работе «___» _____ 2025 г.

Москва 2025

СОДЕРЖАНИЕ

ПРАКТИЧЕСКАЯ РАБОТА 1

1. Цель работы

Освоить базовый синтаксис Kotlin: работу с условными конструкциями (when), циклами (for), массивами, вводом/выводом и простыми алгоритмами (подсчёт символов, размен денег).

2. Решение

2.1.Задание 1

ДНК состоит из 4 типов нуклеотидов: A (аденин), T (тимин), G (гуанин), C (цитозин). Ваша программа получает на вход строку вида ATGCCTCTCTC и должна посчитать число нуклеотидов каждого типа (вывести числа через пробел в порядке как в строке выше).

Для решения задачи подсчёта нуклеотидов была применена встроенная функция readln() для получения строки от пользователя. Для хранения количества нуклеотидов каждого типа был создан массив фиксированного размера с помощью Array(4) {0}. Перебор символов входной строки осуществлялся через цикл for, внутри которого конструкция when позволяла сопоставить каждый символ с соответствующим типом нуклеотида и увеличить соответствующий счётчик в массиве. Результат выводился с помощью print() в виде последовательности чисел, разделённых пробелами.

Код реализации алгоритма программы представлен в листинге 1.1

Листинг 1.1 – Код программы

```
package pract_1

import java.util.Scanner
import kotlin.text.iterator

fun main(){
    //#1
    println("task 1")

    val str = readln()
    var ATGC = Array(4, {0})

    for (x in str) {
        when(x){
            'A' -> ATGC[0]++
            'T' -> ATGC[1]++
            'G' -> ATGC[2]++
            'C' -> ATGC[3]++
        }
```

```
    }

    for (i in ATGC){
        print(" $i")
    }
}
```

Результат работы программы представлен на рисунке 1.1

```
task 1
GTCAGCATCTGTCAGT
 3 5 4 4
```

Рисунок 1.1 – Результат выполнения программы

2.2.Задание 2

В банкомате остались только купюры номиналом 8 4 2 1. Дано положительное число n - количество денег для размена. Необходимо найти минимальное количество купюр, с помощью которых можно разменять это количество денег (соблюсти порядок: первым числом вывести количество купюр номиналом 8, вторым - 4 и т д).

При решении задачи размена денег использовался класс Scanner для чтения целочисленного значения с клавиатуры. Основная логика базировалась на последовательном применении операций целочисленного деления (/) и взятия остатка (%) к исходной сумме, начиная с наибольшего номинала. Количество купюр каждого номинала сохранялось в массиве Array(4) {0}, после чего элементы массива выводились через цикл for в требуемом порядке.

Код реализации алгоритма программы представлен в листинге 1.2

Листинг 1.2 – Код программы

```
package pract_1

import java.util.Scanner
import kotlin.text.iterator

fun main(){
    //#2
    println("\n\ntask 2")

    var count = Array(4, {0})
    val scan = Scanner(System.`in`)
    var x = scan.nextInt()

    count[0] = x / 8
```

```
    x %= 8
    count[1] = x / 4
    x %= 4
    count[2] = x / 2
    count[3] = x % 2

    for (i in count){
        print(" $i")
    }
}
```

Результат работы программы представлен на рисунке 1.2

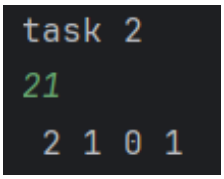

Рисунок 1.2 – Результат выполнения программы

3. Вывод

В ходе выполнения работы были освоены основы Kotlin: работа с массивами, условными операторами, циклами и вводом/выводом. Решены задачи на обработку строк и алгоритмические задачи на размен денег, что позволило закрепить базовые навыки программирования на Kotlin.

ПРАКТИЧЕСКАЯ РАБОТА 2

1. Цель работы

Изучить работу с массивами и строками в Kotlin: ввод массивов, подсчёт среднего значения, анализ повторяющихся элементов, поиск дубликатов.

2. Решение

2.1. Задание 1

На вход подается число N — длина массива. Затем передается массив вещественных чисел (ai) из N элементов. Необходимо подсчитать среднее арифметическое всех чисел массива. Вывести среднее арифметическое на экран.

Была реализована программа для расчёта среднего арифметического. С помощью Scanner считывались размер массива и его элементы, которые сохранялись в массиве типа DoubleArray. Для вычисления среднего значения использовался встроенный метод average(), что позволило избежать ручного суммирования.

Код реализации алгоритма программы представлен в листинге 2.1

Листинг 2.1 – Код программы

```
package pract_2

import java.util.Scanner

fun main(){
    val scan = Scanner(System.`in`)

    //#1
    println("task #1")

    val N = scan.nextInt()
    val numbers = DoubleArray(N)

    for (i in 0..N-1) {
        numbers[i] = scan.nextDouble()
    }

    val average = numbers.average()
    print(average)
}
```

Результат работы программы представлен на рисунке 2.1

```
task #1
5
56 23 65 3 89
47.2
```

Рисунок 2.1 – Результат выполнения программы

2.2.Задание 2

На вход подается число N — длина массива. Затем передается массив целых чисел (*a*i) из N элементов, отсортированный по возрастанию. Необходимо вывести на экран построчно сколько встретилось различных элементов. Каждая строка должна содержать количество элементов и сам элемент через пробел.

Для подсчёта повторяющихся элементов в отсортированном массиве применялся подход с двумя вложенными циклами while. Внешний цикл проходил по всем элементам, а внутренний подсчитывал количество одинаковых подряд идущих значений, используя переменную-счётчик. Как только встречался новый элемент, программа выводила текущий результат и переходила к следующему уникальному значению.

Код реализации алгоритма программы представлен в листинге 2.2

Листинг 2.2 – Код программы

```
package pract_2

import java.util.Scanner

fun main(){
    val scan = Scanner(System.`in`)

    //#2
    println("\n\ntask #2")

    val n = scan.nextInt()
    val arr = IntArray(n)

    for (i in 0..n-1) {
        arr[i] = scan.nextInt()
    }

    var i = 0
    while (i < n){
        val current = arr[i]
        var count = 0

        while (i < n && arr[i] == current){
            count++
            i++
```

```
        }

        println("$count $current")
    }
}
```

Результат работы программы представлен на рисунке 2.2

```
task #2
8
1 1 2 2 2 2 6 8
2 1
4 2
1 6
1 8
```

Рисунок 2.2 – Результат выполнения программы

2.3.Задание 3

На вход подается число N — длина массива. Затем передается массив строк из N элементов (разделение через перевод строки). Каждая строка содержит только строчные символы латинского алфавита. Необходимо найти и вывести дубликат на экран. Гарантируется что он есть и только один.

Поиск дубликата в массиве строк выполнялся методом прямого попарного сравнения всех элементов с помощью двух вложенных циклов for. При обнаружении совпадения строка выводилась на экран, и выполнение функции завершалось досрочно с помощью return.

Код реализации алгоритма программы представлен в листинге 2.3

Листинг 2.3 – Код программы

```
package pract_2

import java.util.Scanner

fun main(){
    val scan = Scanner(System.`in`)

    //#3
    println("\n\ntask #3")

    val Nn = scan.nextInt()
    var strings = Array(Nn) { readLine() }

    for (i in 0..Nn - 1) {
        for (j in i + 1..Nn-1) {
            if (strings[i] == strings[j]) {
```

```
                println(strings[i])
                return
            }
        }
    }
}
```

Результат работы программы представлен на рисунке 2.3

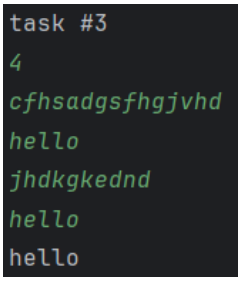

Рисунок 2.3 – Результат выполнения программы

3. Вывод

В ходе работы были изучены методы обработки массивов и строк, включая подсчёт статистики, анализ повторяющихся элементов и поиск дубликатов. Приобретены навыки работы с индексами и алгоритмами обработки последовательностей.

ПРАКТИЧЕСКАЯ РАБОТА 3

1. Цель работы

Освоить работу с функциями, случайными числами, строковыми преобразованиями и генерацией паролей.

2. Решение

2.1.Задание 1

Необходимо реализовать игру. Алгоритм игры должен быть записан в отдельной функции. В функции main должен быть только вызов функции с алгоритмом игры.

Условие следующее:

Компьютер «загадывает» (с помощью генератора случайных чисел) целое число M в промежутке от 0 до 1000 включительно. Затем предлагает пользователю угадать это число. Пользователь вводит число с клавиатуры. Если пользователь угадал число M, то вывести на экран "Победа!". Если введенное пользователем число меньше M, то вывести на экран "Это число меньше загаданного."

Если введенное число больше, то вывести "Это число больше загаданного." Продолжать игру до тех пор, пока число не будет отгадано или пока не будет введено любое отрицательное число.

Реализовать консольное меню.

Была разработана игра "Угадай число". Логика игры была инкапсулирована в отдельную функцию game(). Случайное число генерировалось с помощью Random.nextInt(). В цикле while пользователь вводил свои варианты, а с помощью цепочки условных операторов if-else программа сравнивала введённое число с загаданным и выдавала соответствующие подсказки. Цикл прерывался либо при угадывании, либо при вводе отрицательного числа.

Код реализации алгоритма программы представлен в листинге 3.1

Листинг 3.1 – Код программы

```
package pract_3

import java.util.Scanner
import kotlin.random.Random
import kotlin.text.iterator

val scan = Scanner(System.`in`)

fun game() {
    val number = Random.nextInt(0, 1001)
    println(number)
    println("Я загадал число от 0 до 1000. Вам нужно угадать его")
    var choice = scan.nextInt()

    while (choice >= 0) {
        if (choice == number) {
            println("Победа!")
            break
        }
        else if (choice < number) {
            println("Это число меньше загаданного")
            choice = scan.nextInt()
        }
        else {
            println("Это число больше загаданного")
            choice = scan.nextInt()
        }
    }
}

fun main() {
    game()
}
```

Результат работы программы представлен на рисунке 3.1

```
Я загадал число от 0 до 1000. Вам нужно угадать его
50
Это число больше загаданного
10
Это число меньше загаданного
16
Победа!
```

Рисунок 3.1 – Результат выполнения программы

2.2.Задание 2

На вход подается строка S, состоящая только из русских заглавных букв (без Ё). Необходимо реализовать функцию, которая кодирует переданную строку с помощью азбуки Морзе и затем вывести результат на экран. Отделять коды

букв нужно пробелом. Необходимо использовать функции. В функции main должен быть вызов функции с реализацией алгоритма.

Массив с кодами Морзе: {".-", "-...", ".--", "--.", "-..", ".", "...-", "--..", "..", ".---", "-.-", ".-..", "--", "-.", "---", ".--.", ".-.", "...", "-", "..-", "..-.", "....", "-.-.", "---.", "----", "--.-", "--.--", "-.--", "-..-", "..-..", "..--", ".-.-"}.

Для кодирования строки в азбуку Морзе была создана функция code_Morse(). В ней использовался цикл for для перебора символов строки и конструкция when, сопоставлявшая каждый русский символ с соответствующей последовательностью точек и тире. Для эффективного построения итоговой строки применялся StringBuilder. После обработки всех символов лишний пробел в конце удалялся методом trim().

Код реализации алгоритма программы представлен в листинге 3.2

Листинг 3.2 – Код программы

```
package pract_3

import java.util.Scanner
import kotlin.random.Random
import kotlin.text.iterator

val scan = Scanner(System.`in`)

fun code_Morse(str: String): String {
    val new_str = StringBuilder()
    for (symbol in str) {
        when (symbol) {
            'А' -> new_str.append(".-")
            'Б' -> new_str.append("-...")
            'В' -> new_str.append(".--")
            'Г' -> new_str.append("--.")
            'Д' -> new_str.append("-..")
            'Е' -> new_str.append(".")
            'Ж' -> new_str.append("...-")
            'З' -> new_str.append("--..")
            'И' -> new_str.append("..")
            'Й' -> new_str.append(".---")
            'К' -> new_str.append("-.-")
            'Л' -> new_str.append(".-..")
            'М' -> new_str.append("--")
            'Н' -> new_str.append("-.")
            'О' -> new_str.append("---")
            'П' -> new_str.append(".--.")
            'Р' -> new_str.append(".-.")
            'С' -> new_str.append("...")
            'Т' -> new_str.append("-")
            'У' -> new_str.append("..-")
            'Ф' -> new_str.append("..-.")
            'Х' -> new_str.append("....")
            'Ц' -> new_str.append("-.-.")
            'Ч' -> new_str.append("---.")
```

```
            'Ш' -> new_str.append("----")
            'Щ' -> new_str.append("--.-")
            'Ъ' -> new_str.append("--.--")
            'Ы' -> new_str.append("-.--")
            'Ь' -> new_str.append("-..-")
            'Э' -> new_str.append("..-..")
            'Ю' -> new_str.append("..--")
            'Я' -> new_str.append(".-.-")
        }
        new_str.append(" ")
    }
    return new_str.toString().trim()
}

fun main() {
    println("\n\nВведите строку русских букв:")
    val str = readln()
    println(code_Morse(str))
}
```

Результат работы программы представлен на рисунке 3.2

```
Введите строку русских букв:
ПРИВЕТ
.--. .-. .. .-- . -
```

Рисунок 3.2 – Результат выполнения программы

2.3.Доп. задание

Создать программу, генерирующую пароль.

Необходимо использовать функции. В функции main должен быть вызов функции с реализацией алгоритма.

На вход подается число N — длина желаемого пароля. Необходимо проверить, что N >= 8, иначе вывести на экран "Пароль с N количеством символов небезопасен" (подставить вместо N число) и предложить пользователю еще раз ввести число N. Если N >= 8 то сгенерировать пароль, удовлетворяющий условиям ниже и вывести его на экран. В пароле должны быть:

- заглавные латинские символы
- строчные латинские символы
- числа
- специальные знаки (_*-)

Была реализована генерация безопасного пароля. В функции create_password() с помощью цикла while и условного оператора if проверялось, чтобы длина пароля была не менее 8 символов. Основная логика генерации находилась в функции generate_password(), где из наборов символов (заглавные, строчные, цифры, специальные) с помощью Random.nextInt() сначала выбирался минимум по одному символу из каждой категории для гарантии наличия, а затем оставшаяся длина пароля заполнялась случайными символами из общего пула. Итоговый пароль дополнительно перемешивался методами shuffled() и joinToString().

Код реализации алгоритма программы представлен в листинге 3.3

Листинг 3.3 – Код программы

```
package pract_3

import java.util.Scanner
import kotlin.random.Random
import kotlin.text.iterator

val scan = Scanner(System.`in`)

fun create_password() {
    print("\n\nВведите длину пароля (от 8 символов): ")
    var len = scan.nextInt()

    while (len < 8) {
        println("Пароль с $len количеством символов небезопасен")
        print("Введите длину пароля (от 8 символов): ")
        len = scan.nextInt()
    }

    print("Сгенерированный пароль: ")
    println(generate_password(len))
}


fun generate_password(len: Int): String {
    val uppercase = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    val lowercase = "abcdefghijklmnopqrstuvwxyz"
    val digits = "0123456789"
    val special = "!@#$%^&*()-_=+[]{}|;:,.<>?/"
    val allChars = uppercase + lowercase + digits + special

    val password = StringBuilder()

    password.append(uppercase[Random.nextInt(uppercase.length)])
    password.append(lowercase[Random.nextInt(lowercase.length)])
    password.append(digits[Random.nextInt(digits.length)])
    password.append(special[Random.nextInt(special.length)])

    for (i in 1..len - 4) {
        password.append(allChars[Random.nextInt(allChars.length)])
    }
```

```
    return password.toList().shuffled().joinToString("")
}


fun main() {
    create_password()
}
```

Результат работы программы представлен на рисунке 3.3

```
Введите длину пароля (от 8 символов): 13
Сгенерированный пароль: xZv7Z=ZUjR(CU
```

Рисунок 3.3 – Результат выполнения программы

3. Вывод

Были изучены принципы работы с функциями, случайными числами и строковыми операциями. Реализованы алгоритмы игры, кодирования Морзе и генерации паролей, что позволило углубить понимание структурного программирования на Kotlin.

ПРАКТИЧЕСКАЯ РАБОТА 4

1. Цель работы

Освоить разработку интерактивных консольных приложений, работу с коллекциями и анонимными функциями.

2. Решение

2.1.Задание 1

Реализуйте консольную версию игры «Текстовый квест» с одним-двумя поворотами сюжета. Описание:

1. Программа описывает игроку ситуацию и предлагает варианты действий (пронумерованные).
2. Игрок вводит номер выбранного действия.
3. В зависимости от выбора, ситуация меняется, и предлагаются новые варианты.
4. Игра продолжается до одной из нескольких концовок.

Разработан текстовый квест с использованием принципа рекурсивных вызовов функций. Навигация по сюжету организовывалась через систему вложенных функций, каждая из которых представляла собой локацию или выбор. Для обработки решений пользователя применялись анонимные функции (лямбда-выражения), принимающие номер выбора и через конструкцию when определяющие следующее действие. Такая архитектура позволила создать разветвлённый сюжет с несколькими концовками.

Код реализации алгоритма программы представлен в листинге 4.1

Листинг 4.1 – Код программы

```
package pract_4

import kotlin.system.exitProcess

fun main() {
    println("=== ТЕКСТОВЫЙ КВЕСТ: ПОБЕГ ИЗ ЛАБИРИНТА ===")
    startGame()
}

fun startGame() {
    fun showMenu() {
        println("\n=== ГЛАВНОЕ МЕНЮ ===")
        println("1. Начать игру")
        println("2. Правила игры")
        println("3. Выход")
```

```
        print("Выберите вариант: ")
    }

    fun handleChoice(choice: Int) {
        when (choice) {
            1 -> beginAdventure()
            2 -> showRules()
            3 -> {
                println("До свидания!")
                exitProcess(0)
            }
            else -> println("Неверный выбор. Попробуйте снова.")
        }
    }

    while (true) {
        showMenu()
        val choice = readln().toInt()
        handleChoice(choice)
    }
}

fun showRules() {
    println("\n=== ПРАВИЛА ИГРЫ ===")
    println("• Выбирайте варианты действий, вводя соответствующие цифры")
    println("• Принимайте решения осторожно - они влияют на концовку")
    println("• Цель: найти выход из лабиринта")
    println("\nНажмите Enter чтобы продолжить...")
    readln()
}

fun beginAdventure() {
    println("\n" + "=".repeat(50))
    println("Вы просыпаетесь в темном каменном помещении.")
    println("Вокруг тишина, лишь капает вода где-то вдалеке.")
    println("Перед вами три прохода...")

    firstChoice()
}

fun firstChoice() {
    val options = listOf(
        "Пойти в левый проход",
        "Пойти в центральный проход",
        "Пойти в правый проход"
    )

    val handleFirstChoice: (Int) -> Unit = { choice ->
        when (choice) {
            1 -> leftPath()
            2 -> centerPath()
            3 -> rightPath()
            else -> {
                println("Неверный выбор.")
                firstChoice()
            }
        }
    }

    showOptions(options, handleFirstChoice)
}

fun leftPath() {
    println("\nВы идете по левому проходу.")
```

```
    println("Туннель становится уже, воздух спертый.")
    println("Впереди замечаете слабый свет...")

    val options = listOf(
        "Ускорить шаг и двигаться к свету",
        "Вернуться назад",
        "Осмотреться внимательнее"
    )

    val handleLeftChoice: (Int) -> Unit = { choice ->
        when (choice) {
            1 -> {
                println("\nВы вышли к выходу! Это был правильный путь!")
                println("Поздравляем с успешным побегом!")
                showEndMenu()
            }
            2 -> {
                println("\nВы возвращаетесь к началу.")
                firstChoice()
            }
            3 -> {
                println("\nПри внимательном осмотре вы замечаете скрытую
дверь.")
                println("Она ведет к сокровищнице! Альтернативная победа!")
                showEndMenu()
            }
            else -> leftPath()
        }
    }

    showOptions(options, handleLeftChoice)
}

fun centerPath() {
    println("\nЦентральный проход ведет в большой зал.")
    println("В центре зала стоит древний каменный стол.")
    println("На столе лежит старая книга и загадочный ключ...")

    val options = listOf(
        "Взять ключ и продолжить путь",
        "Прочитать книгу",
        "Вернуться назад"
    )

    val handleCenterChoice: (Int) -> Unit = { choice ->
        when (choice) {
            1 -> {
                println("\nС ключом в руке вы находите запертую дверь.")
                println("Ключ подходит! Вы на свободе!")
                showEndMenu()
            }
            2 -> {
                println("\nВ книге написаны древние заклинания.")
                println("Вы случайно активируете магический портал!")
                println("Вас затягивает в неизвестность...")
                showEndMenu()
            }
            3 -> {
                println("\nВы возвращаетесь к началу.")
                firstChoice()
            }
            else -> centerPath()
        }
    }
```

```
    showOptions(options, handleCenterChoice)
}

fun rightPath() {
    println("\nПравый проход ведет вниз по крутой лестнице.")
    println("Внизу слышатся странные звуки...")
    println("Лестница заканчивается перед подземной рекой.")

    val options = listOf(
        "Попытаться перейти реку вброд",
        "Поискать другой путь",
        "Вернуться назад"
    )

    val handleRightChoice: (Int) -> Unit = { choice ->
        when (choice) {
            1 -> {
                println("\nТечение слишком сильное! Вас сносит водой...")
                println("К сожалению, это конец вашего путешествия.")
                showEndMenu()
            }
            2 -> {
                println("\nВы находите старую лодку и переплываете реку.")
                println("На другом берегу - выход на свободу!")
                showEndMenu()
            }
            3 -> {
                println("\nВы возвращаетесь к началу.")
                firstChoice()
            }
            else -> rightPath()
        }
    }

    showOptions(options, handleRightChoice)
}

fun showOptions(options: List<String>, handler: (Int) -> Unit) {
    println("\nВаши варианты:")
    options.forEachIndexed { index, option ->
        println("${index + 1}. $option")
    }
    print("Выберите действие (1-${options.size}): ")

    val choice = readln().toInt()
    if (choice in 1..options.size) {
        handler(choice)
    } else {
        println("Неверный выбор. Попробуйте снова.")
        showOptions(options, handler)
    }
}

fun showEndMenu() {
    val options = listOf(
        "Начать заново",
        "В главное меню",
        "Выйти из игры"
    )

    val handleEndChoice: (Int) -> Unit = { choice ->
        when (choice) {
            1 -> beginAdventure()
```

```
            2 -> startGame()
            3 -> {
                println("Спасибо за игру!")
                exitProcess(0)
            }
            else -> showEndMenu()
        }
    }

    println("\n" + "=".repeat(30))
    println("ИГРА ЗАВЕРШЕНА")
    showOptions(options, handleEndChoice)
}
```

Результат работы программы представлен на рисунке 4.1

```
=== ГЛАВНОЕ МЕНЮ ===
1. Начать игру
2. Правила игры
3. Выход
Выберите вариант: 1

==================================================
Вы просыпаетесь в темном каменном помещении.
Вокруг тишина, лишь капает вода где-то вдалеке.
Перед вами три прохода...

Ваши варианты:
1. Пойти в левый проход
2. Пойти в центральный проход
3. Пойти в правый проход
Выберите действие (1-3): 1

Вы идете по левому проходу.
Туннель становится уже, воздух спертый.
Впереди замечаете слабый свет...

Ваши варианты:
1. Ускорить шаг и двигаться к свету
2. Вернуться назад
3. Осмотреться внимательнее
Выберите действие (1-3): 1

Вы вышли к выходу! Это был правильный путь!
Поздравляем с успешным побегом!
```

Рисунок 4.1 – Результат выполнения программы

2.2.Задание 2

Реализовать функцию, которая принимает массив words и целое положительное число k.

Необходимо вернуть k наиболее часто встречающихся слов.

Результирующий массив должен быть отсортирован по убыванию частоты встречаемого слова. В случае одинакового количества частоты для слов, то отсортировать и выводить их по убыванию в лексикографическом порядке.

Для поиска наиболее частых слов была реализована функция topKFrequent(). В ней использовалась изменяемая карта mutableMapOf для подсчёта частоты встречаемости каждого слова. Затем записи карты преобразовывались в список и сортировались с помощью метода sortedWith с кастомным компаратором. Этот компаратор сначала сравнивал частоту слов (по убыванию), а при равенстве частот — сами слова в лексикографическом порядке (также по убыванию). Итоговый список из k элементов формировался с помощью цикла for.

Код реализации алгоритма программы представлен в листинге 4.2

Листинг 4.2 – Код программы

```
package pract_4

fun topKFrequent(words: Array<String>, k: Int): List<String> {
    val frequency = mutableMapOf<String, Int>()
    for (word in words) {
        val currentCount = frequency[word] ?: 0
        frequency[word] = currentCount + 1
    }

    val wordFreqList = frequency.toList()

    val sorted = wordFreqList.sortedWith { a, b ->
        if (a.second == b.second) {
            b.first.compareTo(a.first)
        } else {
            b.second.compareTo(a.second)
        }
    }

    val result = mutableListOf<String>()
    for (i in 0 .. k) {
        if (i < sorted.size) {
            result.add(sorted[i].first)
        }
    }
    return result
}
```

```
fun main() {
    val words =
arrayOf("the","day","is","sunny","the","the","the","sunny","is","is","day")
    println(topKFrequent(words, 4)) // [the, is, day, sunny]
}
```

Результат работы программы представлен на рисунке 4.2

```
[the, is, sunny, day]
```

Рисунок 4.2 – Результат выполнения программы

3. Вывод

В ходе работы были изучены принципы построения интерактивных консольных приложений, работа с анонимными функциями и коллекциями. Реализованы текстовый квест и алгоритм поиска наиболее частых слов, что позволило развить навыки проектирования пользовательских интерфейсов и обработки данных.

ПРАКТИЧЕСКАЯ РАБОТА 5

1. Цель работы

Освоить основы объектно-ориентированного программирования в Kotlin: создание классов, инкапсуляцию, свойства, методы, сервисные классы.

2. Решение

2.1.Задание 1

Создайте класс с именем Cat.

В классе должны быть следующие приватные функции:

- rest() - выводит на экран «Sleep»
- voice() - выводит на экран «Meow»
- feed() - выводит на экран «Eat»

Также необходимо реализовать одну публичную функцию:

- randomAction() - случайным образом вызывает одну из закрытых функцию.

Был создан класс Cat, демонстрирующий инкапсуляцию. Три приватных метода (rest(), voice(), feed()) выводили соответствующие действия кота. Единственный публичный метод randomAction() с помощью Random.nextInt() и конструкции when случайным образом вызывал один из приватных методов, имитируя поведение объекта.

Код реализации алгоритма программы представлен в листинге 5.1

Листинг 5.1 – Код программы

```
package pract_5
import kotlin.random.Random

class Cat {
    private fun rest() {
        println("Sleep")
    }

    private fun voice() {
        println("Meow")
    }

    private fun feed() {
        println("Eat")
    }

    public fun randomAction() {
        val action = Random.nextInt(0, 3)
        when (action) {
```

```
            0 -> this.rest()
            1 -> this.voice()
            2 -> this.feed()
        }
    }
}

package pract_5

fun main() {
    println("task_1")

    val cat = Cat()
    cat.randomAction()
    cat.randomAction()
    cat.randomAction()
}
```

Результат работы программы представлен на рисунке 5.1

```
task_1
Meow
Sleep
Eat
```

Рисунок 5.1 – Результат выполнения программы

2.2.Задание 2

Создайте класс Student.

В этом классе должны быть следующие приватные свойства:

- var firstName: String - имя студента. Геттер для этого свойства должен возвращать значение с первой заглавной буквой, а сеттер - убирать лишние пробелы при установке значения.
- var lastName: String - фамилия студента. Геттер для этого свойства также должен возвращать значение с первой заглавной буквой, а сеттер - убирать лишние пробелы при установке значения.
- var scores: IntArray - массив из последних десяти оценок студента

А также следующие публичные методы:

- Методы для получения и изменения свойства firstName с реализацией описанной логики

- Методы для получения и изменения свойства lastName с аналогичной логикой
- Методы для получения и изменения массива scores
- Метод, который добавляет новую оценку в массив scores, удаляя первую оценку и добавляя новую в конец (например: 2, 5, 4 → добавляем 3 → результат: 5, 4, 3). Реализовать именно через массив
- Метод, возвращающий среднюю оценку студента (считается как среднее арифметическое всех элементов массива scores)

В классе Student были реализованы кастомные геттеры и сеттеры. Для свойств firstName и lastName сеттер удалял лишние пробелы с помощью trim(), а геттер возвращал строку с первой заглавной буквой через replaceFirstChar. Массив оценок scores обрабатывался с помощью методов addScore() (сдвигающего старые оценки и добавляющего новую) и getAverageScore() (использующего average() для расчёта среднего балла).

Код реализации алгоритма программы представлен в листинге 5.2

Листинг 5.2 – Код программы

```
package pract_5

class Student {
    private var firstName: String = ""
        get() = field.replaceFirstChar { it.uppercase() }
        set(value) { field = value.trim() }
    private var lastName: String = ""
        get() = field.replaceFirstChar { it.uppercase() }
        set(value) { field = value.trim() }
    private var scores: IntArray = IntArray(10)

    fun setfirstName(value: String) {
        this.firstName = value
    }

    fun getfirstName(): String {
        return this.firstName
    }

    fun setlastName(value: String) {
        this.lastName = value
    }

    fun getlastName(): String {
        return this.lastName
    }

    fun getScores(): IntArray = scores.copyOf()
```

```
    fun setScores(newScores: IntArray) {
        scores = newScores.copyOf()
    }

    fun addScore(newScore: Int) {
        for (i in 0..scores.size - 2) {
            scores[i] = scores[i + 1]
        }
        scores[scores.size - 1] = newScore
    }

    fun getAverageScore(): Double {
        return if (scores.isEmpty()) 0.0
        else scores.average()
    }
}

fun main() {
    println("\n\ntask_2")

    val student = Student()

    student.setfirstName("  Nikita  ")
    student.setlastName(" pavlov")
    student.setScores(intArrayOf(5, 4, 3, 2, 2, 3, 4, 5, 5, 5))

    student.addScore(4)

    println(student.getfirstName())
    println(student.getlastName())
    println(student.getScores().contentToString())
    println(student.getAverageScore())
}
```

Результат работы программы представлен на рисунке 5.2

```
task_2
Nikita
Pavlov
[4, 3, 2, 2, 3, 4, 5, 5, 5, 4]
3.7
```

Рисунок 5.2 – Результат выполнения программы

2.3.Задание 3

Создайте класс StudentService.

В классе должны быть следующие публичные функции:

- findBestStudent() - принимает на вход массив объектов типа Student (из предыдущего задания), возвращает студента с наивысшей средней оценкой. Если таких студентов несколько, вернуть любого.

- sortStudentsByLastName() - принимает массив объектов Student, сортирует его по фамилиям в алфавитном порядке и возвращает отсортированный массив.

Был создан сервисный класс StudentService. Метод findBestStudent() перебирал массив студентов, сравнивая их средний балл и запоминая студента с максимальным значением. Метод sortStudentsByLastName() использовал встроенную функцию sortedBy для сортировки массива по фамилии студента в алфавитном порядке.

Код реализации алгоритма программы представлен в листинге 5.3

Листинг 5.3 – Код программы

```
package pract_5

class StudentService {
    fun findBestStudent(students: Array<Student>): Student? {
        if (students.isEmpty()) return null

        var bestStudent = students[0]
        var maxAverage = bestStudent.getAverageScore()

        for (i in 1..students.size - 1) {
            val currentAverage = students[i].getAverageScore()
            if (currentAverage > maxAverage) {
                maxAverage = currentAverage
                bestStudent = students[i]
            }
        }

        return bestStudent
    }

    fun sortStudentsByLastName(students: Array<Student>): Array<Student> {
        return students.sortedBy { it.getlastName() }.toTypedArray()
    }
}

fun main() {
    println("\n\ntask_3")

    val student1 = Student().apply {
        setfirstName("иван")
        setlastName("петров")
        setScores(intArrayOf(5, 5, 5, 5, 5, 5, 5, 5, 5, 5)) // средний: 5.0
    }

    val student2 = Student().apply {
        setfirstName("анна")
        setlastName("сидорова")
        setScores(intArrayOf(4, 4, 4, 4, 4, 4, 4, 4, 4, 4)) // средний: 4.0
    }

    val student3 = Student().apply {
        setfirstName("петр")
        setlastName("иванов")
```

```
        setScores(intArrayOf(5, 4, 5, 4, 5, 4, 5, 4, 5, 4)) // средний: 4.5
    }

    val students = arrayOf(student1, student2, student3)
    val studentService = StudentService()

    val bestStudent = studentService.findBestStudent(students)
    println("Лучший студент: ${bestStudent?.getfirstName()}
${bestStudent?.getlastName()}")
    println("Средний балл: ${bestStudent?.getAverageScore()}")

    val sortedStudents = studentService.sortStudentsByLastName(students)
    println("\nСтуденты отсортированные по фамилии:")
    sortedStudents.forEach {
        println("${it.getlastName()} ${it.getfirstName()}")
    }
}
```

Результат работы программы представлен на рисунке 5.3

```
task_3
Лучший студент: Иван Петров
Средний балл: 5.0

Студенты отсортированные по фамилии:
Иванов Петр
Петров Иван
Сидорова Анна
```

Рисунок 5.3 – Результат выполнения программы

3. Вывод

Были изучены основные принципы ООП в Kotlin: создание классов, инкапсуляция данных, работа со свойствами и методами. Реализованы классы Cat, Student и StudentService, что позволило понять организацию данных и логики в объектно-ориентированном стиле.

ПРАКТИЧЕСКАЯ РАБОТА 6

1. Цель работы

Освоить создание классов с конструкторами, валидацией данных, методами для работы со временем и строками, а также разработку многофайловых проектов.

2. Решение

2.1.Задание 1

Необходимо реализовать класс TimeMeasure с функционалом, описанным ниже (необходимые поля продумать самостоятельно). Обязательно должны быть реализованы валидации на входные параметры.

Конструкторы:

- Возможность создать TimeMeasure, задав часы, минуты и секунды.
- Возможность создать TimeMeasure, задав часы и минуты. Секунды тогда должны проставиться нулевыми.
- Возможность создать TimeMeasure, задав часы. Минуты и секунды тогда должны проставиться нулевыми.

Публичные методы:

- Вывести на экран установленное в классе время в формате hh:mm:ss
- Вывести на экран установленное в классе время в 12-часовом формате (используя hh:mm:ss am/pm)
- Метод, который прибавляет переданное время к установленному в TimeMeasure (на вход передаются только часы, минуты и секунды).

Класс TimeMeasure был реализован с несколькими конструкторами, позволяющими задавать время с разной степенью детализации (часы, минуты и секунды; только часы и минуты; только часы). Проверка корректности вводимых значений выполнялась в приватной функции validateTime(), которая выбрасывала исключение IllegalArgumentException при выходе за допустимые диапазоны. Для вывода времени использовалось форматирование строк через String.format() с заполнением нулями. Метод addTime() корректно складывал интервалы времени с учётом переноса секунд в минуты, а минут в часы.

Код реализации алгоритма программы представлен в листинге 6.1

Листинг 6.1 – Код программы

```
package pract_6

class TimeMeasure {
    private var hours: Int = 0
    private var minutes: Int = 0
    private var seconds: Int = 0


    constructor(hours: Int, minutes: Int, seconds: Int) {
        validateTime(hours, minutes, seconds)
        this.hours = hours
        this.minutes = minutes
        this.seconds = seconds
    }


    constructor(hours: Int, minutes: Int): this(hours, minutes, 0)
    constructor(hours: Int): this(hours, 0, 0)


    private fun validateTime(hours: Int, minutes: Int, seconds: Int) {
        if (hours < 0 || hours > 23) {
            throw IllegalArgumentException("Часы должны быть в диапазоне 0-
23")
        }
        if (minutes < 0 || minutes > 59) {
            throw IllegalArgumentException("Минуты должны быть в диапазоне
0-59")
        }
        if (seconds < 0 || seconds > 59) {
            throw IllegalArgumentException("Секунды должны быть в диапазоне
0-59")
        }
    }


    fun display24HourFormat() {
        println(String.format("%02d:%02d:%02d", hours, minutes, seconds))
    }


    fun display12HourFormat() {
        val period = if (hours < 12) "am" else "pm"
        val newHours = when {
            hours == 0 -> 12
            hours > 12 -> hours - 12
            else -> hours
        }
        println(String.format("%02d:%02d:%02d %s", newHours, minutes,
seconds, period))
    }


    fun addTime(hours: Int, minutes: Int, seconds: Int) {
        if (hours < 0 || minutes < 0 || seconds < 0) {
            throw IllegalArgumentException("Добавляемое время не может быть
отрицательным")
        }
        var totalSeconds = this.seconds + seconds
        var totalMinutes = this.minutes + minutes + totalSeconds / 60
```

```
        var totalHours = this.hours + hours + totalMinutes / 60
        this.seconds = totalSeconds % 60
        this.minutes = totalMinutes % 60
        this.hours = totalHours % 24
    }

}

fun main() {
    println("task_1")

    val time1 = TimeMeasure(12, 34, 17)
    time1.display24HourFormat()
    time1.addTime(1, 1, 1)
    time1.display12HourFormat()
    println()

    val time2 = TimeMeasure(23, 17)
    time2.display24HourFormat()
    time2.display12HourFormat()
    println()

    val time3 = TimeMeasure(8)
    time3.display24HourFormat()
    time3.display12HourFormat()
}
```

Результат работы программы представлен на рисунке 6.1

```
task_1
12:34:17
01:35:18 pm

23:17:00
11:17:00 pm

08:00:00
08:00:00 am
```

Рисунок 6.1 – Результат выполнения программы

2.2.Задание 2

Необходимо реализовать класс UniqueString, который хранит внутри себя строку как массив char и предоставляет следующий функционал:

Конструкторы:

- Создание UniqueString, принимая на вход массив char
- Создание UniqueString, принимая на вход String

Публичные методы (названия методов, входные и выходные параметры продумать самостоятельно):

- Вернуть i-ый символ строки
- Вернуть длину строки
- Вывести строку на экран
- Проверить, есть ли переданная подстрока в UniqueString (на вход подается массив char). Вернуть true, если найдена и false иначе
- Проверить, есть ли переданная подстрока в UniqueString (на вход подается String). Вернуть true, если найдена и false иначе
- Удалить из строки UniqueString ведущие пробельные символы, если они есть
- Развернуть строку (первый символ должен стать последним, а последний первым и т.д.)

Класс UniqueString хранил строку как массив символов CharArray. Были реализованы методы для базовых операций: доступ к символу по индексу с проверкой границ, получение длины, вывод на экран. Поиск подстроки выполнялся вручную с помощью двух вложенных циклов, перебирающих возможные позиции начала подстроки. Метод trimStart() находил первый не пробельный символ и возвращал новый объект UniqueString с частью исходного массива. Метод reverse() создавал новый массив символов, заполняя его в обратном порядке.

Код реализации алгоритма программы представлен в листинге 6.2

Листинг 6.2 – Код программы

```
package pract_6

class UniqueString {
    private var charArray: CharArray


    constructor(chars: CharArray) {
        this.charArray = chars.copyOf()
    }


    constructor(str: String) {
        this.charArray = str.toCharArray()
    }
```

```
    fun getCharAt(index: Int): Char {
        if (index < 0 || index >= charArray.size) {
            throw IndexOutOfBoundsException("Индекс $index выходит за
границы строки (0..${charArray.size - 1})")
        }
        return charArray[index]
    }


    fun length(): Int {
        return charArray.size
    }


    fun print() {
        println(String(charArray))
    }


    fun contains(substring: CharArray): Boolean {
        if (substring.isEmpty()) return true
        if (substring.size > charArray.size) return false
        for (i in 0..charArray.size - substring.size) {
            var found = true
            for (j in 0..substring.size - 1) {
                if (charArray[i + j] != substring[j]) {
                    found = false
                    break
                }
            }
            if (found) return true
        }
        return false
    }


    fun contains(substring: String): Boolean {
        return contains(substring.toCharArray())
    }


    fun trimStart(): UniqueString {
        var startIndex = 0
        while (startIndex < charArray.size && charArray[startIndex] == ' ')
{
            startIndex++
        }
        return if (startIndex == 0) {
            this
        } else {
            UniqueString(charArray.copyOfRange(startIndex, charArray.size))
        }
    }


    fun reverse(): UniqueString {
        val reversed = CharArray(charArray.size)
        for (i in 0..charArray.size - 1) {
            reversed[i] = charArray[charArray.size - 1 - i]
        }
        return UniqueString(reversed)
```

```
    }
}

fun main() {
    println("\n\ntask_2")

    val str1 = UniqueString(charArrayOf(' ', ' ', 'H', 'e', 'l', 'l', 'o', '
', 'w', 'o', 'r', 'l', 'd'))
    val str2 = UniqueString("  Hello world")

    println(str1.getCharAt(2))
    println(str2.getCharAt(2))

    println(str1.length())
    println(str2.length())

    str1.print()
    str2.print()

    println(str1.contains(charArrayOf('w', 'o', 'r', 'l', 'd')))
    println(str2.contains("world"))

    val str1clean = str1.trimStart()
    val str2clean = str2.trimStart()

    str1clean.reverse().print()
    str2clean.reverse().print()
}
```

Результат работы программы представлен на рисунке 6.2

```
task_2
H
H
13
13
  Hello world
  Hello world
true
true
dlrow olleH
dlrow olleH
```

Рисунок 6.2 – Результат выполнения программы

2.3.Доп. задание

Реализовать класс «банкомат» Atm.

Создайте класс BankAccount в пакете banking

Реализуйте основной конструктор с параметрами: accountNumber, initialBalance, ownerName

Добавьте вторичный конструктор, принимающий только accountNumber и ownerName (баланс по умолчанию = 0)

Свойства и модификаторы:

- accountNumber - только для чтения (val), public
- balance - private, с кастомным геттером, возвращающим округленное значение
- ownerName - private с кастомным сеттером, проверяющим длину имени (не менее 2 символов)
- transactionCount - private, счетчик операций

Методы:

- Пополнение счета (Double),
- Снятие средств (проверять достаточность баланса),
- Возвращение форматированной строки с информацией о счете.

Особенности:

- Прямой доступ к балансу извне запрещен,
- Все изменения баланса только через методы,
- Необходимо добавить private метод logTransaction() для увеличения счетчика операций
- Также, добавить проверку начального баланса (не может быть отрицательным) через блок инициализации.

Проект был организован в виде отдельного пакета banking. В классе BankAccount использовался блок init для проверки начального баланса. Были реализованы кастомные аксессоры: для свойства balance геттер округлял значение до двух знаков, для ownerName сеттер проверял длину имени. Приватный метод logTransaction() увеличивал счётчик операций при каждом изменении баланса. Класс Atm управлял коллекцией счетов, предоставляя методы для их создания, поиска, пополнения, снятия средств и вывода информации.

Код реализации алгоритма программы представлен в листингах 6.3 – 6.5

Листинг 6.3 – Класс BankAccount

```
package pract_6.banking

class BankAccount(
    val accountNumber: String,
    initialBalance: Double,
    ownerName: String
) {
    var balance: Double
        get() = Math.round(field * 100.0) / 100.0
    var ownerName: String = ""
        set(value) {
            if (value.length < 2) {
                throw IllegalArgumentException("Имя владельца должно
содержать не менее 2 символов")
            }
            field = value
        }
    private var transactionCount: Int = 0


    constructor(accountNumber: String, ownerName: String) :
this(accountNumber, 0.0, ownerName)


    init {
        if (initialBalance < 0) {
            throw IllegalArgumentException("Начальный баланс не может быть
отрицательным")
        }
        this.balance = initialBalance
        this.ownerName = ownerName
    }


    fun deposit(amount: Double) {
        if (amount <= 0) {
            throw IllegalArgumentException("Сумма пополнения должна быть
положительной")
        }
        balance += amount
        logTransaction()
    }


    fun withdraw(amount: Double): Boolean {
        if (amount <= 0) {
            throw IllegalArgumentException("Сумма снятия должна быть
положительной")
        }
        if (balance >= amount) {
            balance -= amount
            logTransaction()
            return true
        }
        return false
    }


    fun getAccountInfo(): String {
        return "Счет: $accountNumber\n" +
```

```
                "Владелец: $ownerName\n" +
                "Баланс: ${"%.2f".format(balance)} руб.\n" +
                "Количество операций: $transactionCount"
    }


    private fun logTransaction() {
        transactionCount++
    }
}
```

Листинг 6.4 – Класс Atm

```
package pract_6.banking

class Atm {
    private val accounts = mutableListOf<BankAccount>()


    fun createAccount(accountNumber: String, ownerName: String,
initialBalance: Double = 0.0) {
        val account = BankAccount(accountNumber, initialBalance, ownerName)
        accounts.add(account)
    }


    fun findAccount(accountNumber: String): BankAccount? {
        return accounts.find { it.accountNumber == accountNumber }
    }


    fun depositToAccount(accountNumber: String, amount: Double): Boolean {
        val account = findAccount(accountNumber)
        return if (account != null) {
            try {
                account.deposit(amount)
                true
            } catch (e: IllegalArgumentException) {
                false
            }
        } else {
            false
        }
    }


    fun withdrawFromAccount(accountNumber: String, amount: Double): Boolean
{
        val account = findAccount(accountNumber)
        return account?.withdraw(amount) ?: false
    }


    fun getAccountInfo(accountNumber: String): String? {
        val account = findAccount(accountNumber)
        return account?.getAccountInfo()
    }


    fun getAllAccountsInfo(): String {
        if (accounts.isEmpty()) {
            return "Нет зарегистрированных счетов"
        }
        return accounts.joinToString("\n\n") { it.getAccountInfo() }
```

```
    }
}
```

Листинг 6.5 – Функция main

```
package pract_6
import pract_6.banking.Atm

fun main() {
    println("\n\ntask_bank")

    val atm = Atm()

    try {
        val account1 = atm.createAccount("1234567890", "Иван Иванов",
1000.0)
        val account2 = atm.createAccount("0987654321", "Петр Петров")

        println("1. Информация о счетах:")
        println(atm.getAccountInfo("1234567890"))
        println("\n" + atm.getAccountInfo("0987654321"))

        println("\n2. Пополнение счета:")
        atm.depositToAccount("0987654321", 500.75)
        println("После пополнения 500.75 руб:")
        println(atm.getAccountInfo("0987654321"))

        println("\n3. Снятие средств:")
        val success1 = atm.withdrawFromAccount("1234567890", 300.50)
        println("Снятие 300.50 руб со счета 1234567890: ${if (success1)
"Успешно" else "Недостаточно средств"}")

        val success2 = atm.withdrawFromAccount("1234567890", 2000.0)
        println("Снятие 2000 руб со счета 1234567890: ${if (success2)
"Успешно" else "Недостаточно средств"}")

        println("\n4. Информация после операций:")
        println(atm.getAccountInfo("1234567890"))

        println("\n5. Все счета в банкомате:")
        println(atm.getAllAccountsInfo())

    } catch (e: Exception) {
        println("Произошла ошибка: ${e.message}")
    }
}
```

Результат работы программы представлен на рисунке 6.3

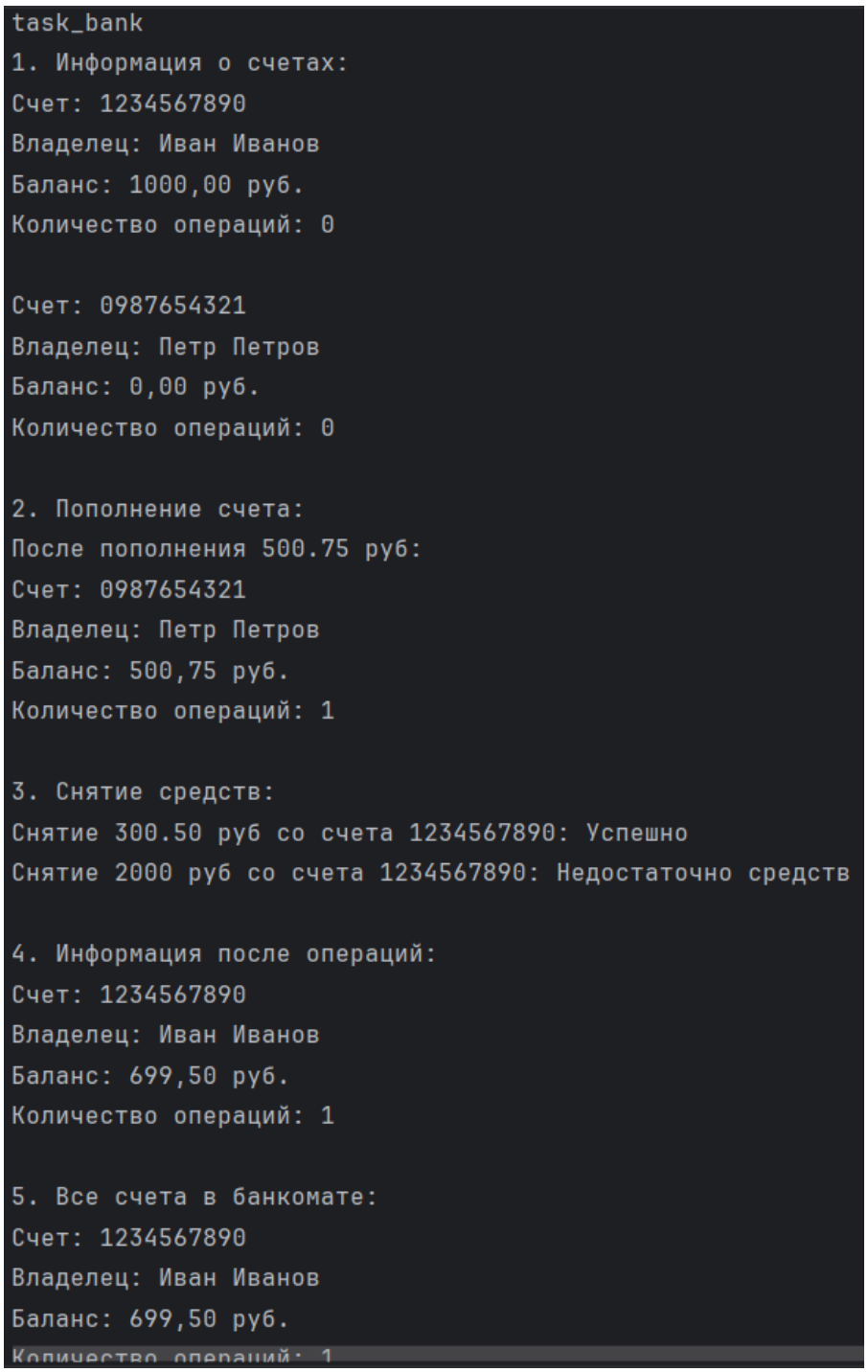

Рисунок 6.3 – Результат выполнения программы

3. Вывод

В ходе работы изучены принципы проектирования классов с валидацией, работа со временем и строками, а также организация многофайловых проектов. Реализованы классы TimeMeasure, UniqueString и система банковских счетов, что позволило углубить понимание ООП и архитектуры приложений.

ПРАКТИЧЕСКАЯ РАБОТА 7

1. **Цель работы**

Освоить принципы наследования, абстрактных классов, data-классов и перечислений в Kotlin.

2. **Решение**

Реализуйте иерархию классов для системы формирования заказов в кафе.

Часть 1.

1. Создайте абстрактный класс MenuItem.
2. Объявите абстрактное свойство name (название).
3. Объявите абстрактное свойство basePrice (базовая цена).
4. Объявите абстрактный метод calculateFinalPrice(), который будет вычислять итоговую цену с учетом всех надбавок (например, за размер порции).
5. Свойство id (уникальный идентификатор) должно быть задано в базовом классе как val id: String = UUID.randomUUID().toString(). Запретите переопределение этого свойства.

Часть 2.

1. Создайте data-класс Ingredient с полями name и isAllergen (является ли аллергеном).

Часть 3.

1. Создайте класс Drink, наследующийся от MenuItem.
2. Переопределите свойства name и basePrice.
3. Добавьте свойство size (тип Size). Создайте enum class Size { SMALL, MEDIUM, LARGE }.
4. Переопределите метод calculateFinalPrice(). Логика: SMALL - basePrice * 1.0, MEDIUM - basePrice * 1.5, LARGE - basePrice * 2.0.
5. Запретите дальнейшее переопределение метода calculateFinalPrice() в классах-потомках (если они будут).

Часть 4.

1. Создайте класс Food, наследующийся от MenuItem.

2. Переопределите свойства name и basePrice.
3. Добавьте свойство ingredients: List, содержащее список ингредиентов.
4. Добавьте свойство isVegetarian. Реализуйте его кастомный геттер, который проверяет, что все ингредиенты не являются аллергенами (просто пример логики, в реальности это не так). Геттер должен возвращать true, если в списке ingredients нет ни одного ингредиента с isAllergen = true.
5. Переопределите метод calculateFinalPrice(). Итоговая цена равна basePrice.

Для реализации иерархии классов системы заказов был создан абстрактный класс MenuItem, определяющий общие свойства (name, basePrice) и абстрактный метод calculateFinalPrice(). Уникальный идентификатор id генерировался в базовом классе с помощью UUID.randomUUID() и был помечен как val, что запрещало его переопределение. Для хранения информации об ингредиентах использовался data-класс Ingredient.

Класс Drink наследовался от MenuItem и добавлял свойство size типа Size (перечисление). Метод calculateFinalPrice() был реализован с помощью when, который умножал базовую цену на коэффициент в зависимости от размера, и помечен как final, чтобы запретить дальнейшее переопределение.

Класс Food также наследовался от MenuItem и содержал список ингредиентов. Кастомный геттер isVegetarian использовал функцию none для проверки, что ни один ингредиент не является аллергеном (в учебных целях). Метод calculateFinalPrice() в этом классе просто возвращал базовую цену.

Код реализации алгоритма программы представлен в листингах 7.1 – 7.6

Листинг 7.1 – Класс MenuItem

```
package pract_7

import java.util.UUID

abstract class MenuItem {
    val id: String = UUID.randomUUID().toString()
    abstract val name: String
    abstract val basePrice: Double
```

```
    abstract fun calculateFinalPrice(): Double
}
```

Листинг 7.2 – Data-класс Ingredient

```
package pract_7

data class Ingredient(
    val name: String,
    val isAllergen: Boolean
)
```

Листинг 7.3 – Класс Drink

```
package pract_7

class Drink(
    override val name: String,
    override val basePrice: Double,
    val size: Size
): MenuItem() {
    final override fun calculateFinalPrice(): Double {
        return when (size) {
            Size.SMALL -> basePrice * 1.0
            Size.MEDIUM -> basePrice * 1.5
            Size.LARGE -> basePrice * 2.0
        }
    }
}
```

Листинг 7.4 – Enum-касс Size

```
package pract_7

enum class Size {
    SMALL, MEDIUM, LARGE
}
```

Листинг 7.5 – Класс Food

```
package pract_7

class Food(
    override val name: String,
    override val basePrice: Double,
    val ingredients: List<Ingredient>
): MenuItem() {
    val isVegetarian: Boolean
        get() = ingredients.none { it.isAllergen }

    override fun calculateFinalPrice(): Double {
        return basePrice
    }
}
```

Листинг 7.6 – Функция main

```
package pract_7

fun main() {
    val coffee = Drink("Капучино", 150.0, Size.MEDIUM)
    println("Напиток: ${coffee.name}, Цена: ${coffee.calculateFinalPrice()}
руб.")

    val salad = Food(
        "Цезарь",
        300.0,
        listOf(
```

```
            Ingredient("Салат", false),
            Ingredient("Курица", false),
            Ingredient("Соус", true)
        )
    )
    println("Блюдо: ${salad.name}, Вегетарианское: ${salad.isVegetarian},
Цена: ${salad.calculateFinalPrice()} руб.")
}
```

Результат работы программы представлен на рисунке 7.1

```
Напиток: Капучино, Цена: 225.0 руб.
Блюдо: Цезарь, Вегетарианское: false, Цена: 300.0 руб.
```

Рисунок 7.1 – Результат выполнения программы

3. Вывод

Были изучены принципы наследования, абстрактные классы, data-классы и перечисления. Реализована иерархия классов для системы заказов в кафе, что позволило понять организацию сложных структур данных и полиморфизм в Kotlin.

ПРАКТИЧЕСКАЯ РАБОТА 8

1. Цель работы

Освоить работу с enum-классами, их свойствами и методами, а также научиться находить и исправлять ошибки в коде на Kotlin.

2. Решение

2.1.Задание 1

Создайте enum класс для представления типов напитков. Каждый тип напитка должен иметь свойство, указывающее на его объем (например, в миллилитрах). Добавьте метод, который возвращает название напитка в формате с заглавной буквы, метод возвращения объема напитка. Добавьте метод isHot(), который возвращает true если температура > 60. Добавьте несколько напитков и выведите информацию о каждом напитке, включая его название и объем.

Был создан enum-класс DrinkType, который принимал в первичном конструкторе параметры volume и temperature. Для каждого элемента перечисления переопределялся абстрактный метод getFormattedName(), возвращающий читаемое название напитка. Были добавлены методы getVolume(), возвращающий объём, и isHot(), проверяющий, превышает ли температура 60 градусов. Метод getDrinkInfo() формировал информационную строку, используя все доступные свойства.

Код реализации алгоритма программы представлен в листингах 8.1 – 8.2

Листинг 8.1 – Enum-класс DrinkType

```
enum class DrinkType(
    private val volume: Int,
    val temperature: Int
) {
    COFFEE(250, 85) {
        override fun getFormattedName(): String = "Кофе"
    },
    TEA(200, 75) {
        override fun getFormattedName(): String = "Чай"
    },
    JUICE(300, 5) {
        override fun getFormattedName(): String = "Сок"
    },
    WATER(500, 10) {
        override fun getFormattedName(): String = "Вода"
    };

    abstract fun getFormattedName(): String
```

```
    fun getVolume(): Int = this.volume

    fun isHot(): Boolean = this.temperature > 60

    fun getDrinkInfo(): String {
        val heatStatus = if (isHot()) "горячий" else "холодный"
        return "${getFormattedName()} - ${getVolume()} мл, $heatStatus
($temperature°C)"
    }
}
```

Листинг 8.2 – Функция main

```
package pract_8

fun main() {
    val coffee = DrinkType.COFFEE
    println("Полная информация о напитке:")
    println(coffee.getDrinkInfo())

    println("\nИнформация об отдельных параметрах:")
    println("Название: ${coffee.getFormattedName()}")
    println("Объем: ${coffee.getVolume()} мл")
    println("Горячий: ${coffee.isHot()}")
    println("Температура: ${coffee.temperature}°C")


    val tea = DrinkType.TEA
    println("\n\nПолная информация о напитке:")
    println(tea.getDrinkInfo())

    println("\nИнформация об отдельных параметрах:")
    println("Название: ${tea.getFormattedName()}")
    println("Объем: ${tea.getVolume()} мл")
    println("Горячий: ${tea.isHot()}")
    println("Температура: ${tea.temperature}°C")
}
```

Результат работы программы представлен на рисунке 8.1

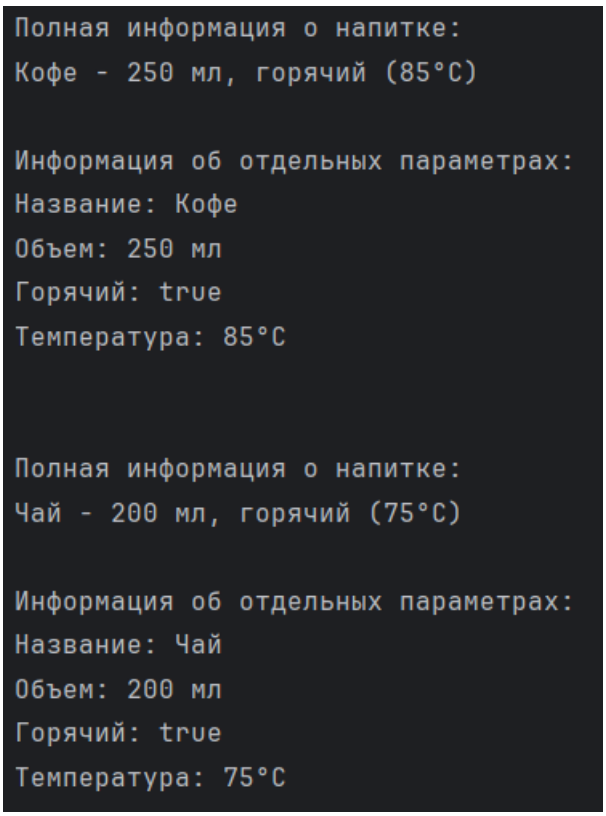

Рисунок 8.1 – Результат выполнения программы

2.2.Задание 2

2.1 Исправьте ошибки в данном коде: https://clck.ru/3Q7qHg.

2.2. Найдите и исправьте ошибки:

```
open class Animal {
        fun speak() = "Some sound"
}

class Cat : Animal()
         override fun speak() = "Meow!"
}

fun main() {
        val cat = Cat()
        println(cat.speak())
}
```

2.3. Найдите и исправьте ошибки:

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

В первой части задания были проанализированы и исправлены ошибки в предоставленном коде банковской системы. Исправления включали: удаление лишних модификаторов open, добавление ключевого слова override для переопределяемых методов, исправление логики обновления баланса (не присваивание, а изменение), корректную инициализацию коллекций, использование valueOf для преобразования строк в enum. Во второй части было добавлено ключевое слово open к методу в родительском классе и override в классе-наследнике. В третьей части к свойствам data-класса были добавлены модификаторы val для корректной генерации компонентов.

Код реализации алгоритма программы представлен в листингах 8.3 – 8.5

Листинг 8.3 – Часть 1

```
import java.util.UUID


abstract class BankCard(val cardNumber: String, var pinCode: Int) {
    abstract fun getBalance(): Double
    abstract fun updateBalance(amount: Double)
}


class CreditCard(cardNumber: String, pinCode: Int, val creditLimit: Double)
: BankCard(cardNumber, pinCode) {
    private var debt: Double = 0.0

    override fun getBalance(): Double {
        return creditLimit - debt // Баланс = кредитный лимит - долг
    }

    override fun updateBalance(amount: Double) {
        debt -= amount // При пополнении карты долг уменьшается
    }

    // Функционал метода такой же, как и у получения баланса
    /*fun getAvailableCredit(): Double { // Удален open, так как класс никем
не наследуется и метод не переопределяется
        return creditLimit - debt
    }*/
}


class DebitCard(cardNumber: String, pinCode: Int) : BankCard(cardNumber,
pinCode) {
    private var balance: Double = 0.0

    override fun getBalance(): Double {
        return balance
    }
```

```
    override fun updateBalance(amount: Double) {
        balance += amount // Увеличиваем\уменьшаем баланс, а не заменяем его
    }


    data class AdditionalInfo(val ownerName: String) // Не имеет смысл
использовать data class с одним свойством
    // лучше просто добавить свойство ownerName в класс BankCard
}


enum class TransactionType {
    WITHDRAWAL,
    DEPOSIT // лишняя ;

    // Функция не имеет смысла внутри класса
    /*fun fromString(type: String): TransactionType {
        return valueOf(type.uppercase())
    }*/
}


data class Transaction(
    val cardNumber: String,
    val amount: Double,
    val date: String,
    val type: TransactionType
) {
    val transactionId: String = UUID.randomUUID().toString() // Генерация
уникальных id
}


class ATM {

    private val transactions: MutableList<Transaction> = mutableListOf() //
Удален метод и добавлена инициализация

    fun makeTransaction(card: BankCard, amount: Double, date: String, type:
TransactionType): Boolean {

        when (type) {
            TransactionType.WITHDRAWAL -> {
                if (card.getBalance() >= amount) {
                    card.updateBalance(-amount)
                    transactions.add(Transaction(card.cardNumber, amount,
date, type))
                    return true
                }
            }
            TransactionType.DEPOSIT -> {
                card.updateBalance(amount)
                transactions.add(Transaction(card.cardNumber, amount, date,
type))
                return true
            }

        }
        return false
    }

    fun printTransactions(cardNumber: String) {
```

```
        val cardTransactions = transactions.filter { it.cardNumber ==
cardNumber }
        println("Транзакции по карте $cardNumber:")
        for (transaction in cardTransactions) {

            println("${transaction.date}: ${transaction.type}
${transaction.amount}")
        }
    }


    fun getAllTransactions(): List<Transaction> { // Метод возвращает
неизменяемый список
        return transactions.toList()
    }
}


fun main() {
    val atm = ATM()

    val creditCard = CreditCard("1234-5678-9012-3456", 1234, 10000.0)
    val debitCard = DebitCard("9876-5432-1098-7654", 5678)

    debitCard.updateBalance(5000.0)

    atm.makeTransaction(creditCard, 2000.0, "2025-01-15",
TransactionType.WITHDRAWAL)
    atm.makeTransaction(debitCard, 1000.0, "2025-01-15",
TransactionType.DEPOSIT) // Передаем тип транзакции, а не строку с названием

    println("Баланс кредитной карты: ${creditCard.getBalance()}")
    println("Баланс дебетовой карты: ${debitCard.getBalance()}")

    atm.printTransactions("1234-5678-9012-3456")
}
```

Листинг 8.4 – Часть 2

```
package pract_8

open class Animal {
    open fun speak() = "Some sound"  // Функция open для переопределения
}
class Cat : Animal() {
    override fun speak() = "Meow!"
}
fun main() {
    val cat = Cat()
    println(cat.speak())
}
```

Листинг 8.5 – Часть 3

```
package pract_8

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

Результат работы программ представлен на рисунках 8.2 – 8.3

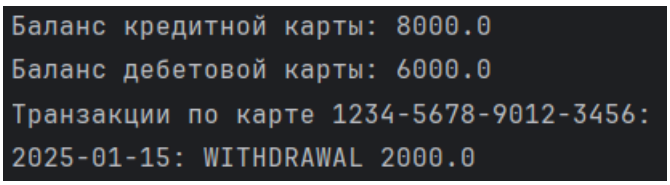

Рисунок 8.2 – Результат выполнения части 1

Рисунок 8.3 – Результат выполнения части 2

3. Вывод

В ходе работы изучены возможности enum-классов в Kotlin, их методы и свойства. Также проведён анализ и исправление типичных ошибок в коде, что позволило улучшить понимание синтаксиса и семантики языка.

ПРАКТИЧЕСКАЯ РАБОТА 9

1. Цель работы

Освоить построение сложных иерархий классов, интерфейсов, работу с динамическими состояниями объектов и систему взаимодействия между ними.

2. Решение

2.1. Задание 1

Имеется следующий набор утверждений.

Рассматриваются следующие животные (можно заменить на другие):

- летучая мышь (Bat)
- дельфин (Dolphin)
- золотая рыбка (GoldFish)
- орел (Eagle)

Все животные одинаково едят и спят (предположим), и никто из животных не должен иметь возможности делать это иначе.

Еще животные умеют по-разному рождаться (wayOfBirth)

- Млекопитающие (Mammal) живородящие.
- Рыбы (Fish) мечут икру.
- Птицы (Bird) откладывают яйца.

Помимо этого, бывают некоторые особенности, касающиеся передвижения. Бывают летающие животные (Flying) и плавающие (Swimming). Однако орел летает быстро, а летучая мышь медленно. Дельфин плавает быстро, а золотая рыбка медленно.

Согласно этим утверждениям, нужно создать иерархию, состоящую из классов, абстрактных классов и/или интерфейсов. Каждое действие или утверждение подразумевает под собой вызов Unit метода, в котором реализован вывод на экран описания текущего действия.

Также, необходимо реализовать систему «настроения» животных. Каждое животное должно иметь динамические показатели:

- Голод (0-100%) - увеличивается со временем
- Энергия (0-100%) - уменьшается при активности

- Счастье (0-100%) - зависит от общего состояния

Реализовать методы для управления состоянием животных:

Базовые действия (для всех животных):

- eat() - покормить (уменьшает голод)
- sleep() - уложить спать (восстанавливает энергию)
- play() - поиграть (увеличивает счастье)

Можно, при желании, добавить особые действия.

Реализовать функциональное меню (просмотр животных, взаимодействие с каждым животным, состояние животного -> обновление состояния и т.п), протестировать созданный функционал и показать преподавателю.

Была построена сложная иерархия классов, основанная на абстрактных классах Animal, Mammal, Fish, Bird и интерфейсах Flying, Swimming. Каждый конкретный класс животного (Bat, Dolphin, GoldFish, Eagle) наследовался от соответствующего абстрактного класса и реализовывал необходимый интерфейс передвижения. Все животные имели динамические состояния (голод, энергия, счастье), которые обновлялись в методе updateState(). Базовые методы eat(), sleep(), play() воздействовали на эти состояния. В основной программе было реализовано консольное меню, позволяющее пользователю взаимодействовать с каждым животным, просматривать его состояние и наблюдать за изменением параметров со временем.

Код реализации алгоритма программы представлен в листингах 9.1 – 9.11

Листинг 9.1 – Класс Animal

```
package pract_9.task_1

abstract class Animal(val name: String) {
    protected var hunger: Int = 0
    protected var energy: Int = 100
    protected var happiness: Int = 100

    abstract fun wayOfBirth()
    abstract fun move()

    open fun eat() {
        hunger = maxOf(0, hunger - 60)
        energy = minOf(100, energy + 40)
        println("$name ест. Голод уменьшен.")
    }

    open fun sleep() {
```

```
        energy = minOf(100, energy + 70)
        hunger = minOf(100, hunger + 20)
        println("$name спит. Энергия восстановлена.")
    }

    open fun play() {
        happiness = minOf(100, happiness + 60)
        energy = maxOf(0, energy - 20)
        hunger = minOf(100, hunger + 10)
        println("$name играет. Счастье увеличено.")
    }

    fun getStatus(): String {
        return """
            Состояние $name:
            Голод: $hunger%
            Энергия: $energy%
            Счастье: $happiness%
        """.trimIndent()
    }

    fun updateState() {
        hunger = minOf(100, hunger + 5)
        energy = maxOf(0, energy - 3)
        if (hunger > 70 || energy < 30) {
            happiness = maxOf(0, happiness - 5)
        }
    }
}
```

Листинг 9.2 – Класс Mammal

```
package pract_9.task_1

abstract class Mammal(name: String) : Animal(name) {
    override fun wayOfBirth() {
        println("$name - млекопитающее, рождает живых детенышей")
    }
}
```

Листинг 9.3 – Класс Fish

```
package pract_9.task_1

abstract class Fish(name: String) : Animal(name) {
    override fun wayOfBirth() {
        println("$name - рыба, мечет икру")
    }
}
```

Листинг 9.4 – Класс Bird

```
package pract_9.task_1

abstract class Bird(name: String) : Animal(name) {
    override fun wayOfBirth() {
        println("$name - птица, откладывает яйца")
    }
}
```

Листинг 9.5 – Interface Flying

```
package pract_9.task_1

interface Flying {
    fun fly()
}
```

Листинг 9.6 – Interface Swimming

```
package pract_9.task_1

interface Swimming {
    fun swim()
}
```

Листинг 9.7 – Класс Bat

```
package pract_9.task_1

class Bat : Mammal("Летучая мышь"), Flying {
    override fun move() {
        fly()
    }

    override fun fly() {
        println("Летучая мышь летает медленно")
        energy = maxOf(0, energy - 10)
    }
}
```

Листинг 9.8 – Класс Dolphin

```
package pract_9.task_1

class Dolphin : Mammal("Дельфин"), Swimming {
    override fun move() {
        swim()
    }

    override fun swim() {
        println("Дельфин плавает быстро")
        energy = maxOf(0, energy - 20)
    }
}
```

Листинг 9.9 – Класс GoldFish

```
package pract_9.task_1

class GoldFish : Fish("Золотая рыбка"), Swimming {
    override fun move() {
        swim()
    }

    override fun swim() {
        println("Золотая рыбка плавает медленно")
        energy = maxOf(0, energy - 10)
    }
}
```

Листинг 9.10 – Класс Eagle

```
package pract_9.task_1

class Eagle : Bird("Орел"), Flying {
    override fun move() {
        fly()
    }

    override fun fly() {
        println("Орел летает быстро")
        energy = maxOf(0, energy - 20)
    }
}
```

Листинг 9.11 – Функция main

```
package pract_9.task_1

fun main() {
    val animals = mutableListOf<Animal>(
        Bat(),
        Dolphin(),
        GoldFish(),
        Eagle()
    )

    fun showAllAnimals() {
        println("\nВСЕ ЖИВОТНЫЕ:")
        animals.forEachIndexed { index, animal ->
            println("${index + 1}. ${animal.name}")
        }
    }

    fun showAnimalStatus(index: Int) {
        if (index in 1..animals.size) {
            val animal = animals[index - 1]
            println("\n${animal.name.uppercase()}:")
            println(animal.getStatus())
        } else {
            println("Некорректный номер животного")
        }
    }

    fun interactWithAnimal(index: Int) {
        if (index !in 1..animals.size) {
            println("Некорректный номер животного")
            return
        }

        val animal = animals[index - 1]

        while (true) {
            println("\nВЗАИМОДЕЙСТВИЕ С ${animal.name.uppercase()}:")
            println("1. Покормить")
            println("2. Уложить спать")
            println("3. Поиграть")
            println("4. Узнать способ рождения")
            println("5. Движение")
            println("6. Показать статус")
            println("0. Назад")

            print("Выберите действие: ")
            when (readLine()?.toIntOrNull()) {
                1 -> animal.eat()
                2 -> animal.sleep()
                3 -> animal.play()
                4 -> animal.wayOfBirth()
                5 -> animal.move()
                6 -> println(animal.getStatus())
                0 -> break
                else -> println("Некорректный выбор")
            }

            animal.updateState()
        }
    }

    fun updateAllAnimals() {
```

```
        animals.forEach { it.updateState() }
        println("Состояние всех животных обновлено")
    }

    fun showMainMenu() {
        while (true) {
            println("\nСИСТЕМА УПРАВЛЕНИЯ ЖИВОТНЫМИ:")
            println("1. Показать всех животных")
            println("2. Показать статус животного")
            println("3. Взаимодействовать с животным")
            println("4. Обновить состояние всех животных")
            println("5. Выйти")

            print("Выберите опцию: ")
            when (readLine()?.toIntOrNull()) {
                1 -> showAllAnimals()
                2 -> {
                    showAllAnimals()
                    print("Выберите животное: ")
                    val index = readLine()?.toIntOrNull() ?: 0
                    showAnimalStatus(index)
                }
                3 -> {
                    showAllAnimals()
                    print("Выберите животное: ")
                    val index = readLine()?.toIntOrNull() ?: 0
                    interactWithAnimal(index)
                }
                4 -> updateAllAnimals()
                5 -> {
                    println("До свидания!")
                    return
                }
                else -> println("Некорректный выбор")
            }
        }
    }

    println("Добро пожаловать в систему управления животными!")
    showMainMenu()
}
```

Результат работы программы представлен на рисунке 9.1

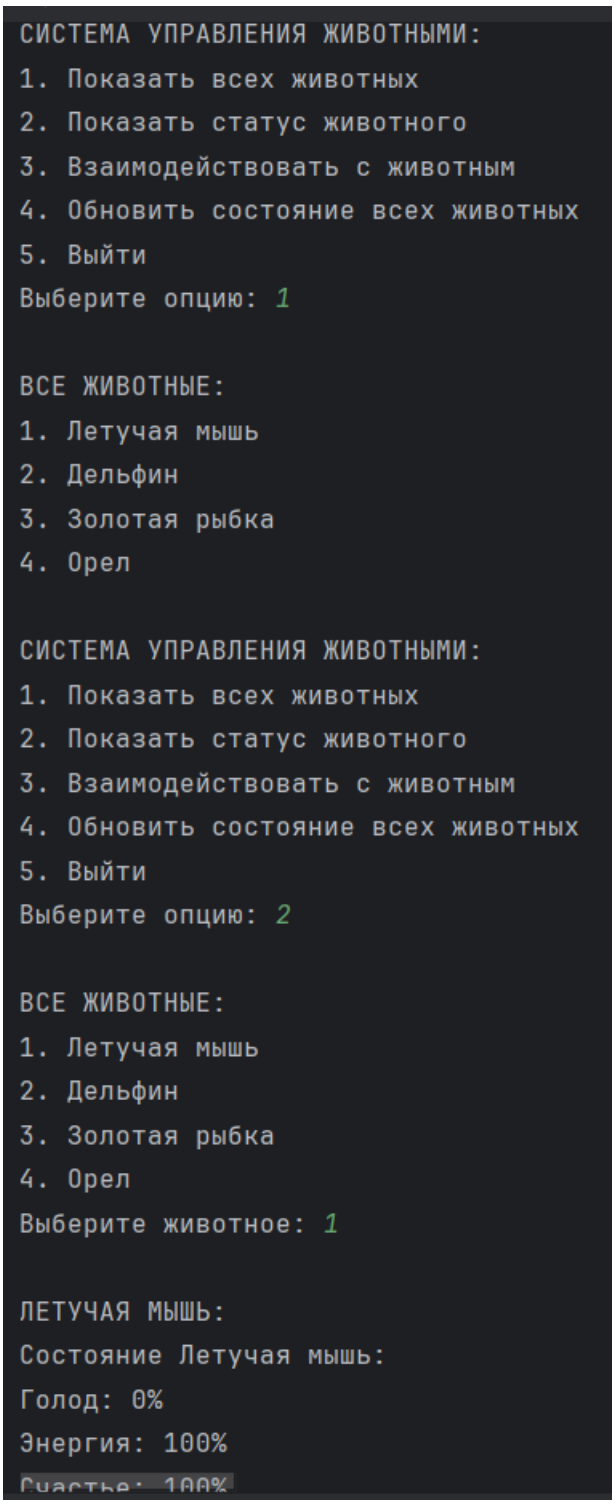

Рисунок 9.1 – Результат выполнения программы

2.2.Задание 2

Мастерская BestRepairEver занимается ремонтом различных видов электроники. Она специализируется на ремонте компьютеров и телефонов. Реализуйте метод в мастерской, который позволяет по переданному устройству

определить, сможет ли мастерская его починить или нет. Метод должен возвращать результат типа boolean. Протестируйте метод.

Была создана система, моделирующая работу ремонтной мастерской. Абстрактный класс ElectronicDevice представлял общий тип электронного устройства. От него наследовались конкретные классы Computer, Phone, Tablet, TV. Функция canRepair() использовала оператор when с проверкой типов через is для определения, может ли мастерская починить переданное устройство. Для компьютеров и телефонов функция возвращала true, для остальных типов — false. Таким образом была продемонстрирована работа с полиморфизмом и проверкой типов во время выполнения.

Код реализации алгоритма программы представлен в листингах 9.12 – 9.17

Листинг 9.12 – Класс ElectronicDevice

```
package pract_9.task_2

abstract class ElectronicDevice(val brand: String, val model: String)
```

Листинг 9.13 – Класс Computer

```
package pract_9.task_2

class Computer(brand: String, model: String, val processor: String) :
ElectronicDevice(brand, model) {
    override fun toString(): String {
        return "Компьютер $brand $model (Процессор: $processor)"
    }
}
```

Листинг 9.14 – Класс Phone

```
package pract_9.task_2

class Phone(brand: String, model: String, val os: String) :
ElectronicDevice(brand, model) {
    override fun toString(): String {
        return "Телефон $brand $model (ОС: $os)"
    }
}
```

Листинг 9.15 – Класс Tablet

```
package pract_9.task_2

class Tablet(brand: String, model: String, val screenSize: Double) :
ElectronicDevice(brand, model) {
    override fun toString(): String {
        return "Планшет $brand $model (Экран: ${screenSize}\")"
    }
}
```

Листинг 9.16 – Класс TV

```
package pract_9.task_2

class TV(brand: String, model: String, val resolution: String) :
ElectronicDevice(brand, model) {
    override fun toString(): String {
        return "Телевизор $brand $model (Разрешение: $resolution)"
    }
}
```

Листинг 9.17 – Функция main

```
package pract_9.task_2

fun canRepair(device: ElectronicDevice): Boolean {
    return when (device) {
        is Computer -> {
            println("Мастерская может починить компьютер: ${device.brand}
${device.model}")
            true
        }
        is Phone -> {
            println("Мастерская может починить телефон: ${device.brand}
${device.model}")
            true
        }
        else -> {
            println("Мастерская НЕ может починить
${device::class.simpleName}: ${device.brand} ${device.model}")
            false
        }
    }
}


fun main() {
    val devices = listOf(
        Computer("HUAWEI", "Matebook D16", "Intel i5"),
        Phone("Samsung", "Galaxy S24", "Android"),
        Computer("Apple", "MacBook Pro", "M3"),
        Phone("Apple", "iPhone 17", "iOS"),
        Tablet("Apple", "iPad Air", 10.9),
        TV("Sony", "Bravia X90J", "4K"),
        Phone("Xiaomi", "Redmi Note 15", "Android"),
        Computer("HP", "Pavilion", "AMD Ryzen 5")
    )

    devices.forEach { device ->
        println("Устройство: $device")
        val canRepair = canRepair(device)
        println("Результат проверки: $canRepair")
        println("---")
    }
}
```

Результат работы программы представлен на рисунке 9.2

```
Устройство: Компьютер HUAWEI Matebook D16 (Процессор: Intel i5)
Мастерская может починить компьютер: HUAWEI Matebook D16
Результат проверки: true
---
Устройство: Телефон Samsung Galaxy S24 (ОС: Android)
Мастерская может починить телефон: Samsung Galaxy S24
Результат проверки: true
---
Устройство: Компьютер Apple MacBook Pro (Процессор: M3)
Мастерская может починить компьютер: Apple MacBook Pro
Результат проверки: true
---
Устройство: Телефон Apple iPhone 17 (ОС: iOS)
Мастерская может починить телефон: Apple iPhone 17
Результат проверки: true
---
Устройство: Планшет Apple iPad Air (Экран: 10.9")
Мастерская НЕ может починить Tablet: Apple iPad Air
Результат проверки: false
---
Устройство: Телевизор Sony Bravia X90J (Разрешение: 4K)
Мастерская НЕ может починить TV: Sony Bravia X90J
Результат проверки: false
---
Устройство: Телефон Xiaomi Redmi Note 15 (ОС: Android)
Мастерская может починить телефон: Xiaomi Redmi Note 15
Результат проверки: true
---
Устройство: Компьютер HP Pavilion (Процессор: AMD Ryzen 5)
Мастерская может починить компьютер: HP Pavilion
Результат проверки: true
---
```

Рисунок 9.2 – Результат выполнения программы

3. Вывод

В ходе работы изучены принципы построения сложных иерархий классов и интерфейсов, работа с динамическими состояниями объектов. Реализованы системы управления животными и ремонта электроники, что позволило углубить понимание полиморфизма и инкапсуляции в ООП.

ПРАКТИЧЕСКАЯ РАБОТА 10

1. Цель работы

Освоить обработку исключений, валидацию данных, работу с анонимными объектами и безопасными вызовами в Kotlin.

2. Решение

2.1.Задание 1

Подготовка анкет для отправки на сервер встретила некоторые препятствия. Так как фронт со своей стороны не сделал проверки, Вася оказался в трудной ситуации, и решил написать собственный класс FormValidator с методами для проверки входных данных перед отправкой анкеты на сервер. Вот описание его задач:

- Метод проверяет, что длина имени составляет от 2 до 20 символов, и первая буква имени заглавная.
- Метод проверяет, что дата рождения находится в диапазоне между 01.01.1900 и текущей датой.
- Метод проверяет, что пол корректно соответствует значениям из перечисления Gender, которое содержит только Male и Female.
- Метод проверяет, что вес является положительным числом и может быть корректно преобразован в тип double.

Помогите Васе, используя классы исключений, корректно обработать анкету перед отправкой.

Был разработан класс FormValidator для комплексной проверки данных анкеты. Метод validateName() проверял длину имени и наличие заглавной первой буквы. Метод validateBirthDate() использовал классы LocalDate и DateTimeFormatter для парсинга и проверки, что дата находится в допустимом диапазоне. Метод validateGender() с помощью valueOf проверял, соответствует ли строка одному из значений enum Gender. Метод validateWeight() пытался преобразовать строку в число и проверял его положительность. В методе validateForm() каждый этап валидации был обёрнут в блок try-catch, что

позволяло собрать все ошибки и вывести их пользователю, не прерывая проверку после первой же ошибки.

Код реализации алгоритма программы представлен в листингах 10.1 – 10.4

Листинг 10.1 – Data-класс FormData

```
package pract_10.task_1

data class FormData(
    val name: String,
    val birthDate: String,
    val gender: String,
    val weight: String
)
```

Листинг 10.2 – Enum-класс Gender

```
package pract_10.task_1

enum class Gender {
    MALE, FEMALE
}
```

Листинг 10.3 – Класс FormValidator

```
package pract_10.task_1

import java.time.LocalDate
import java.time.format.DateTimeFormatter
import java.time.format.DateTimeParseException


class FormValidator {
    fun validateName(name: String) {
        if (name.length < 2 || name.length > 20) {
            throw Exception("Имя должно содержать от 2 до 20 символов")
        }

        if (!name[0].isUpperCase()) {
            throw Exception("Первая буква имени должна быть заглавной")
        }
    }

    fun validateBirthDate(birthDate: String) {
        try {
            val formatter = DateTimeFormatter.ofPattern("dd.MM.yyyy")
            val date = LocalDate.parse(birthDate, formatter)
            val minDate = LocalDate.of(1900, 1, 1)
            val currentDate = LocalDate.now()

            if (date.isBefore(minDate)) {
                throw Exception("Дата рождения не может быть раньше
01.01.1900")
            }

            if (date.isAfter(currentDate)) {
                throw Exception("Дата рождения не может быть в будущем")
            }
        } catch (e: DateTimeParseException) {
            throw Exception("Неверный формат даты. Используйте формат
dd.MM.yyyy")
        }
    }
```

```
    fun validateGender(gender: String) {
        try {
            Gender.valueOf(gender.uppercase())
        } catch (e: IllegalArgumentException) {
            throw Exception("Пол должен быть 'MALE' или 'FEMALE'")
        }
    }

    fun validateWeight(weight: String) {
        try {
            val weightValue = weight.toDouble()
            if (weightValue <= 0) {
                throw Exception("Вес должен быть положительным числом")
            }
        } catch (e: NumberFormatException) {
            throw Exception("Вес должен быть числом")
        }
    }

    fun validateForm(formData: FormData): Boolean {
        val errors = mutableListOf<String>()

        try {
            validateName(formData.name)
        } catch (e: Exception) {
            errors.add("Ошибка в имени: ${e.message}")
        }

        try {
            validateBirthDate(formData.birthDate)
        } catch (e: Exception) {
            errors.add("Ошибка в дате рождения: ${e.message}")
        }

        try {
            validateGender(formData.gender)
        } catch (e: Exception) {
            errors.add("Ошибка в поле: ${e.message}")
        }

        try {
            validateWeight(formData.weight)
        } catch (e: Exception) {
            errors.add("Ошибка в весе: ${e.message}")
        }

        if (errors.isNotEmpty()) {
            println("Обнаружены ошибки валидации:")
            errors.forEach { println("- $it") }
            return false
        }

        println("Анкета прошла валидацию успешно!")
        return true
    }
}
```

Листинг 10.4 – Функция main

```
package pract_10.task_1

fun main() {
    val validator = FormValidator()

    val testForms = listOf(
        FormData("Иван", "15.05.1990", "MALE", "75.5"),
        FormData("иван", "15.05.1990", "MALE", "75.5"),
        FormData("И", "15.05.1990", "MALE", "75.5"),
        FormData("Иван", "15.05.1800", "MALE", "75.5"),
        FormData("Иван", "15.05.1990", "UNKNOWN", "75.5"),
        FormData("Иван", "15.05.1990", "MALE", "-5"),
        FormData("Иван", "15.05.1990", "MALE", "abc")
    )

    testForms.forEachIndexed { index, formData ->
        println("\nТест анкеты ${index + 1}:")
        println("Данные: $formData")
        validator.validateForm(formData)
    }
}
```

Результат работы программы представлен на рисунке 10.1

```
Тест анкеты 1:
Данные: FormData(name=Иван, birthDate=15.05.1990, gender=MALE, weight=75.5)
Анкета прошла валидацию успешно!

Тест анкеты 2:
Данные: FormData(name=иван, birthDate=15.05.1990, gender=MALE, weight=75.5)
Обнаружены ошибки валидации:
- Ошибка в имени: Первая буква имени должна быть заглавной

Тест анкеты 3:
Данные: FormData(name=И, birthDate=15.05.1990, gender=MALE, weight=75.5)
Обнаружены ошибки валидации:
- Ошибка в имени: Имя должно содержать от 2 до 20 символов

Тест анкеты 4:
Данные: FormData(name=Иван, birthDate=15.05.1800, gender=MALE, weight=75.5)
Обнаружены ошибки валидации:
- Ошибка в дате рождения: Дата рождения не может быть раньше 01.01.1900

Тест анкеты 5:
Данные: FormData(name=Иван, birthDate=15.05.1990, gender=UNKNOWN, weight=75.5)
Обнаружены ошибки валидации:
- Ошибка в поле: Пол должен быть 'MALE' или 'FEMALE'

Тест анкеты 6:
Данные: FormData(name=Иван, birthDate=15.05.1990, gender=MALE, weight=-5)
Обнаружены ошибки валидации:
- Ошибка в весе: Вес должен быть положительным числом

Тест анкеты 7:
Данные: FormData(name=Иван, birthDate=15.05.1990, gender=MALE, weight=abc)
Обнаружены ошибки валидации:
- Ошибка в весе: Вес должен быть числом
```

Рисунок 10.1 – Результат выполнения программы

2.2.Задание 2

Напишите программу, которая создает анонимный объект, представляющий собой список пользователей. Каждый пользователь должен иметь имя, возраст и список друзей. Затем программа должна использовать этот

список для выполнения некоторых действий, например, для нахождения самого старшего пользователя в списке и вывода его имени и возраста.

Необходимо создать свой анонимный объект (ключевое слово object), который представляет собой список пользователей, и использовать его для выполнения некоторых действий.

Была продемонстрирована работа с анонимными объектами. Внутри функции main() был создан анонимный объект userDatabase, содержащий список пользователей и несколько методов для его анализа. Метод findOldestUser() с помощью maxByOrNull находил пользователя с максимальным возрастом. Метод findMostSocialUsers() сначала определял максимальное количество друзей, а затем фильтровал пользователей, имеющих такое количество. Метод printStatistics() использовал функции average(), sumOf() и toSet() для вычисления сводных данных. Такой подход позволил инкапсулировать данные и логику их обработки без создания именованного класса.

Код реализации алгоритма программы представлен в листингах 10.5 – 10.6

Листинг 10.5 – Data-класс User

```
package pract_10.task_2

data class User(
    val name: String,
    val age: Int,
    val friends: List<String>
)
```

Листинг 10.6 – Функция main

```
package pract_10.task_2


fun main() {
    val userDatabase = object {
        val users = listOf(
            User("Алексей", 25, listOf("Мария", "Иван", "Елена")),
            User("Мария", 30, listOf("Алексей", "Дмитрий")),
            User("Иван", 35, listOf("Алексей", "Ольга", "Сергей")),
            User("Ольга", 28, listOf("Иван", "Елена")),
            User("Дмитрий", 42, listOf("Мария", "Сергей", "Анна")),
            User("Елена", 38, listOf("Алексей", "Ольга", "Анна")),
            User("Сергей", 45, listOf("Иван", "Дмитрий")),
            User("Анна", 29, listOf("Дмитрий", "Елена"))
        )

        fun findOldestUser(): User? {
            return users.maxByOrNull { it.age }
        }

        fun findMostSocialUsers(): List<User> {
```

```
            val maxFriends = users.maxOfOrNull { it.friends.size } ?: 0
            return users.filter { it.friends.size == maxFriends }
        }

        fun printStatistics() {
            println("\nОбщее количество пользователей: ${users.size}")
            println("Средний возраст: ${users.map { it.age
}.average().toInt()} лет")
            println("Общее количество дружеских связей: ${users.sumOf {
it.friends.size }}")
        }
    }

    val oldestUser = userDatabase.findOldestUser()
    println("\nСамый старший пользователь:")
    if (oldestUser != null) {
        println("Имя: ${oldestUser.name}, Возраст: ${oldestUser.age} лет")
        println("Друзья: ${oldestUser.friends.joinToString(", ")}")
    }

    val mostSocialUsers = userDatabase.findMostSocialUsers()
    println("\nСамые общительные пользователи
(${mostSocialUsers.first().friends.size} друзей):")
    mostSocialUsers.forEach { user ->
        println("- ${user.name}, ${user.age} лет:
${user.friends.joinToString(", ")}")
    }

    userDatabase.printStatistics()
}
```

Результат работы программы представлен на рисунке 10.2

```
Самый старший пользователь:
Имя: Сергей, Возраст: 45 лет
Друзья: Иван, Дмитрий

Самые общительные пользователи (3 друзей):
- Алексей, 25 лет: Мария, Иван, Елена
- Иван, 35 лет: Алексей, Ольга, Сергей
- Дмитрий, 42 лет: Мария, Сергей, Анна
- Елена, 38 лет: Алексей, Ольга, Анна

Общее количество пользователей: 8
Средний возраст: 34 лет
Общее количество дружеских связей: 20
```

Рисунок 10.2 – Результат выполнения программы

3. Вывод

Были изучены методы валидации данных, обработки исключений и работы с анонимными объектами. Реализованы системы проверки анкет и анализа пользовательских данных, что позволило освоить безопасное программирование и организацию данных в Kotlin.

ПРАКТИЧЕСКАЯ РАБОТА 11

1. Цель работы

Освоить scope-функции, обобщённые типы (generics), ограничения типов и расширения функций в Kotlin.

2. Решение

2.1.Задание 1

Разрабатывается система управления заказами в интернет-магазине. Вам предоставлены классы Product и Order. Класс Order содержит информацию о продуктах, стоимости и статусе заказа. Ваша задача – использовать scope функции Kotlin для более удобной работы с объектами заказа (например, инициализация заказа, обработка заказа, расчет скидки и т.п).

Были продемонстрированы возможности scope-функций Kotlin. При создании объекта Order использовалась функция apply, которая позволяла инициализировать свойства объекта в компактной форме. Функция also применялась для побочного эффекта — вывода информации о созданном заказе. Функция let использовалась для вычисления значения с учётом контекста — в данном случае для применения скидки, если общая сумма заказа превышала определённый порог.

Код реализации алгоритма программы представлен в листингах 11.1 – 11.3

Листинг 11.1 – Data-класс Product

```
package pract_11.task_1

data class Product(val name: String, val price: Double)
```

Листинг 11.2 – Класс Order

```
package pract_11.task_1

class Order {
    var products: List<Product> = emptyList()
    var total: Double = 0.0
    var status: String = "Создан"

    override fun toString(): String {
        return "\tПродукты: $products\n\tСтоимость = $total\n\tСтатус =
$status"
    }
}
```

Листинг 11.3 – Функция main

```
package pract_11.task_1

fun main() {
    val order = Order().apply {
        products = listOf(
            Product("Specialized S-Works Tarmac SL8", 17999.25),
            Product("RockShox Sid SL Ultimate", 850.50)
        )
        total = products.sumOf { it.price }
        status = "В сборке"
    }

    order.also {
        println("Заказ создан:\n$it")
    }

    val discount = order.let {
        if (it.total > 1000) it.total * 0.9 else it.total
    }

    println("\tСтоимость с учетом скидки: $discount")
}
```

Результат работы программы представлен на рисунке 11.1

```
Заказ создан:
    Продукты: [Product(name=Specialized S-Works Tarmac SL8, price=17999.25), Product(name=RockShox Sid SL Ultimate, price=850.5)]
    Стоимость = 18849.75
    Статус = В сборке
    Стоимость с учетом скидки: 16964.775
```

Рисунок 11.1 – Результат выполнения программы

2.2.Задание 2

Создайте класс ErrorLogger, который принимает обобщенный тип T и содержит функцию logError, принимающую объект типа T и записывающую его в лог ошибок (для упрощения можно выводить в консоль).

- Если тип T – строка (String), функция должна добавлять к сообщению об ошибке префикс "String error: ".
- Если тип T является числом (Number), функция должна выводить ошибку в формате "Numeric error: [число]".
- Для остальных типов T функция должна записывать сообщение "Unknown error type".

Также создайте функцию расширения для ErrorLogger, которая записывает ошибку и отправляет сообщение на сервер (в данном случае просто выводит сообщение о попытке отправки).

Был создан обобщённый класс ErrorLogger<T>, демонстрирующий параметрический полиморфизм. В методе logError() использовалась конструкция when с проверкой типа через is для определения, какого типа ошибка была передана. Для строк добавлялся префикс "String error:", для чисел — "Numeric error:", для остальных типов выводилось общее сообщение. Также была создана функция-расширение logAndSend(), которая сначала логировала ошибку, а затем выводила сообщение о её отправке на сервер, демонстрируя возможность расширения функциональности существующих классов.

Код реализации алгоритма программы представлен в листингах 11.4 – 11.5

Листинг 11.4 – Класс ErrorLogger

```
package pract_11.task_2

class ErrorLogger<T> {
    fun logError(error: T) {
        when (error) {
            is String -> println("String error: $error")
            is Number -> println("Numeric error: $error")
            else -> println("Unknown error type")
        }
    }
}

fun <T> ErrorLogger<T>.logAndSend(error: T) {
    logError(error)
    println("Отправка ошибки на сервер: $error")
}
```

Листинг 11.5 – Функция main

```
package pract_11.task_2

fun main() {
    val stringLogger = ErrorLogger<String>()
    stringLogger.logError("File not found")

    val numberLogger = ErrorLogger<Number>()
    numberLogger.logError(404)

    val unknownLogger = ErrorLogger<Any>()
    unknownLogger.logError(listOf(1, 2, 3))

    stringLogger.logAndSend("Network timeout")
}
```

Результат работы программы представлен на рисунке 11.2

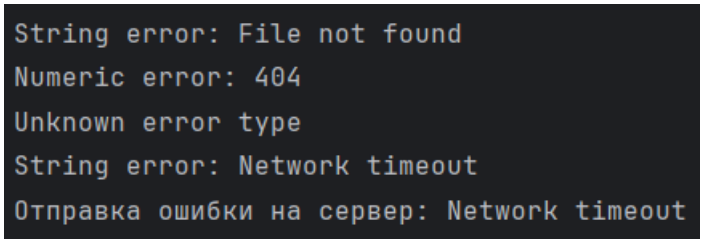

Рисунок 11.2 – Результат выполнения программы

2.3.Задание 3

Напишите класс, который принимает обобщенный тип и выполняет проверку на допустимые значения. Класс должен работать с типами Int, Double и String, проверяя:

1. Для чисел – является ли значение больше нуля.
2. Для строк – длина строки должна быть больше 5 символов.
3. Для других типов – выводить ошибку.

Используйте ограничение обобщений, чтобы тип был либо числом, либо строкой.

Требования:

- Используйте ограничение на типы.
- Реализуйте проверку для разных типов данных.

Был разработан обобщённый класс Validator<T> с ограничением типа <T : Any>, что гарантировало работу только с ненулевыми типами. В методе validate() с помощью конструкции when и проверки типов (is Number, is String) выполнялась соответствующая валидация: для чисел проверялась положительность, для строк — длина. Для типов, не являющихся ни числами, ни строками, выбрасывалось исключение IllegalArgumentException. Такой подход демонстрировал использование обобщений с ограничениями и безопасную работу с разными типами данных.

Код реализации алгоритма программы представлен в листингах 11.6 – 11.7

Листинг 11.6 – Класс Validator

```
package pract_11.task_3

class Validator<T : Any> {
    fun validate(value: T): Boolean {
        return when (value) {
            is Number -> value.toDouble() > 0
            is String -> value.length > 5
            else -> throw IllegalArgumentException("Несоответствие типа")
        }
    }
}
```

Листинг 11.7 – Функция main

```
package pract_11.task_3

fun main() {
    val intValidator = Validator<Int>()
    println("Целочисленное положительное число:
${intValidator.validate(10)}")

    val doubleValidator = Validator<Double>()
    println("Положительное число: ${doubleValidator.validate(34.75)}")

    println("Отрицательное число: ${doubleValidator.validate(-3.52)}")

    val stringValidator = Validator<String>()
    println("Строка (больше 5 элементов):
${stringValidator.validate("HelloWorld")}")

    println("Строка (меньше 5 элементов):
${stringValidator.validate("Hi")}")

    val testValidator = Validator<Char>()
    try {
        println("Буква: ${testValidator.validate('F')}")
    } catch (e: IllegalArgumentException) {
        println("Ошибка: ${e.message}")
    }
}
```

Результат работы программы представлен на рисунке 11.3

```
Целочисленное положительное число: true
Положительное число: true
Отрицательное число: false
Строка (больше 5 элементов): true
Строка (меньше 5 элементов): false
Ошибка: Несоответствие типа
```

Рисунок 11.3 – Результат выполнения программы

2.4.Доп. задание

Для Задания 2 организовать вывод логов не только на консоль, но и в файл.

Была создана расширенная версия логгера — FileErrorLogger<T>, которая не только выводила сообщения в консоль, но и записывала их в файл. Для записи в файл использовался класс File из стандартной библиотеки Java и его метод appendText(). Принцип работы с типами оставался таким же, как в базовом классе, но каждое сообщение дополнительно сохранялось в указанный файл, что демонстрировало сочетание обобщённого программирования и работы с файловой системой.

Код реализации алгоритма программы представлен в листинге 11.8

Листинг 11.8 – Класс FileErrorLogger

```
package pract_11.task_2
import java.io.File

class FileErrorLogger<T>(val logFile: String) {
    fun logError(error: T) {
        val message = when (error) {
            is String -> "String error: $error"
            is Number -> "Numeric error: $error"
            else -> "Unknown error type"
        }
        println(message)
        File(logFile).appendText("$message\n")
    }
}
```

Результат работы программы представлен на рисунке 11.4

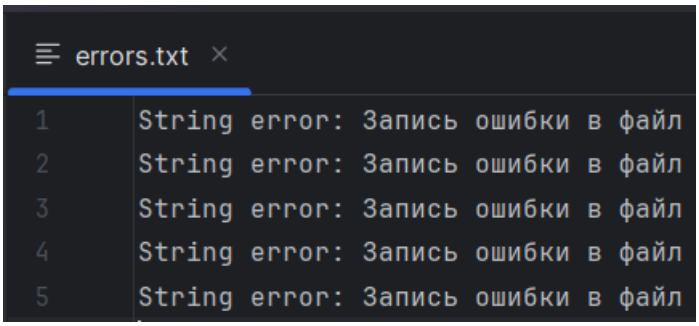

Рисунок 11.4 – Результат выполнения программы

3. Вывод

В ходе работы изучены scope-функции, обобщённые типы, ограничения и расширения функций. Реализованы системы логирования, валидации и работы с файлами, что позволило углубить понимание современных возможностей Kotlin.

ПРАКТИЧЕСКАЯ РАБОТА 12

1. Цель работы

Освоить работу с nullable-типами, безопасными вызовами, расширениями функций и конвертацией единиц измерения.

2. Решение

2.1.Задание 1

Разработайте класс UserProfile, который содержит информацию о пользователе (например, имя, возраст, хобби, любимый цвет, фильм, книга). Некоторые поля могут быть пустыми или null. Напишите метод, который обрабатывает эти значения и выводит информацию о пользователе, используя безопасное обращение к nullable типам.

1. Реализуйте класс UserProfile с nullable свойствами для имени, возраста и т.д.
2. Реализуйте метод printProfileInfo, который будет выводить информацию о пользователе, проверяя поля на null.
3. Добавьте проверку на null для полей и безопасное обращение через операторы ?. и ?:.

Требования:

- Используйте nullable типы.
- Обрабатывайте null значения безопасно.

Был разработан класс UserProfile, все свойства которого были объявлены как nullable-типы, что моделировало ситуацию с неполными данными. В методе printProfileInfo() для каждого свойства использовался оператор безопасного вызова ?. в сочетании с оператором Элвиса ?:. Это позволяло либо выводить фактическое значение свойства, если оно не было null, либо заменять его на строку "Не указано". Такой подход демонстрировал безопасную работу с потенциально отсутствующими данными без риска возникновения NullPointerException.

Код реализации алгоритма программы представлен в листингах 12.1 – 12.2

Листинг 12.1 Класс UserProfile

```
package pract_12.task_1

class UserProfile(
    val name: String?,
    val age: Int?,
    val hobby: String?,
    val favoriteColor: String?,
    val favoriteMovie: String?,
    val favoriteBook: String?
) {
    fun printProfileInfo() {
        println("Профиль пользователя:")
        println("Имя: ${name ?: "Не указано"}")
        println("Возраст: ${age ?: "Не указан"}")
        println("Хобби: ${hobby ?: "Не указано"}")
        println("Любимый цвет: ${favoriteColor ?: "Не указан"}")
        println("Любимый фильм: ${favoriteMovie ?: "Не указан"}")
        println("Любимая книга: ${favoriteBook ?: "Не указана"}")
        println()
    }
}
```

Листинг 12.2 – Функция main

```
package pract_12.task_1

fun main() {
    val user1 = UserProfile(
        name = "Никита",
        age = 20,
        hobby = "Хобби-хорсинг",
        favoriteColor = "Застенчивый паук",
        favoriteMovie = "Зеленый слоник",
        favoriteBook = "Методичка по ОС"
    )

    val user2 = UserProfile(
        name = null,
        age = null,
        hobby = null,
        favoriteColor = null,
        favoriteMovie = null,
        favoriteBook = null
    )

    user1.printProfileInfo()
    user2.printProfileInfo()
}
```

Результат работы программы представлен на рисунке 12.1

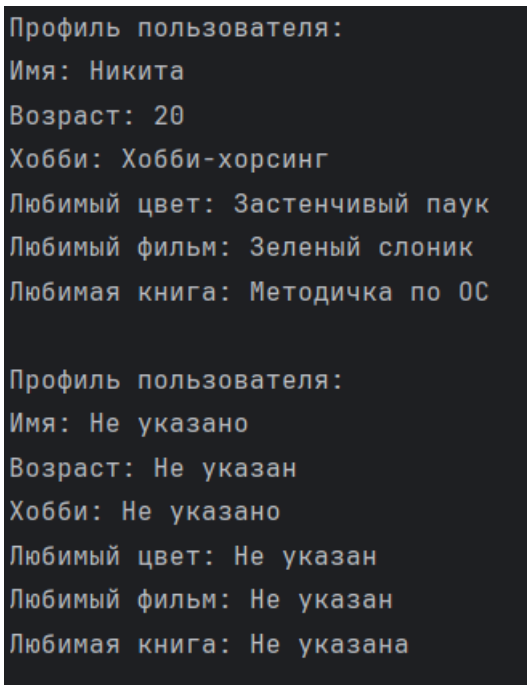

Рисунок 12.1 – Результат выполнения программы

2.2.Задание 2

Напишите программу для конвертации различных единиц измерения (например: температура, расстояние, вес, время, объем, скорость, площадь, энергия, давление, угол, цифровые данные, валюта). Для этого создайте функции расширения, которые будут преобразовывать значения в другие единицы измерения.

1. Создайте функции расширения для типов, которые выполняют преобразования, например:
    a. Конвертируют температуру из Цельсия в Фаренгейты.
    b. Конвертируют километры в мили.
    c. Конвертируют килограммы в фунты.
    d. и другие.
2. Реализовать обработку возможных ошибок ввода.
3. Реализуйте консольное меню для проверки функционала.

Была реализована система конвертации единиц измерения с использованием функций-расширений. Для типа Double были добавлены методы, такие как celsiusToFahrenheit(), kilometersToMiles() и другие, каждый из

которых содержал простую математическую формулу преобразования. Основная программа представляла собой консольное меню, где пользователь выбирал тип преобразования, а затем вводил значение. Для безопасного чтения ввода использовался toDoubleOrNull(), который возвращал null при некорректном вводе, что позволяло обрабатывать ошибки без исключений.

Код реализации алгоритма программы представлен в листинге 12.3

Листинг 12.3 – Код программы

```
package pract_12.task_2

import kotlin.system.exitProcess

fun Double.celsiusToFahrenheit(): Double = this * 9 / 5 + 32
fun Double.fahrenheitToCelsius(): Double = (this - 32) * 5 / 9
fun Double.kilometersToMiles(): Double = this * 0.621371
fun Double.milesToKilometers(): Double = this / 0.621371
fun Double.kilogramsToPounds(): Double = this * 2.20462
fun Double.poundsToKilograms(): Double = this / 2.20462

fun convertTemperatureToFahrenheit() {
    print("Введите температуру в °C: ")
    val celsius = readLine()?.toDoubleOrNull()
    if (celsius != null) {
        println("$celsius °C = ${celsius.celsiusToFahrenheit()} °F\n")
    } else {
        println("Ошибка ввода.\n")
    }
}

fun convertTemperatureToCelsius() {
    print("Введите температуру в °F: ")
    val fahrenheit = readLine()?.toDoubleOrNull()
    if (fahrenheit != null) {
        println("$fahrenheit °F = ${fahrenheit.fahrenheitToCelsius()} °C\n")
    } else {
        println("Ошибка ввода.\n")
    }
}

fun convertKilometersToMiles() {
    print("Введите расстояние в км: ")
    val km = readLine()?.toDoubleOrNull()
    if (km != null) {
        println("$km км = ${km.kilometersToMiles()} миль\n")
    } else {
        println("Ошибка ввода.\n")
    }
}

fun convertMilesToKilometers() {
    print("Введите расстояние в милях: ")
    val miles = readLine()?.toDoubleOrNull()
    if (miles != null) {
        println("$miles миль = ${miles.milesToKilometers()} км\n")
    } else {
        println("Ошибка ввода.\n")
    }
```

```
}

fun convertKilogramsToPounds() {
    print("Введите вес в кг: ")
    val kg = readLine()?.toDoubleOrNull()
    if (kg != null) {
        println("$kg кг = ${kg.kilogramsToPounds()} фунтов\n")
    } else {
        println("Ошибка ввода.\n")
    }
}

fun convertPoundsToKilograms() {
    print("Введите вес в фунтах: ")
    val pounds = readLine()?.toDoubleOrNull()
    if (pounds != null) {
        println("$pounds фунтов = ${pounds.poundsToKilograms()} кг\n")
    } else {
        println("Ошибка ввода.\n")
    }
}

fun main() {
    while (true) {
        println("Конвертер единиц:")
        println("1. Температура: Цельсий → Фаренгейт")
        println("2. Температура: Фаренгейт → Цельсий")
        println("3. Расстояние: Километры → Мили")
        println("4. Расстояние: Мили → Километры")
        println("5. Вес: Килограммы → Фунты")
        println("6. Вес: Фунты → Килограммы")
        println("0. Выход")
        print("Выберите опцию: ")

        when (readLine()?.toIntOrNull()) {
            1 -> convertTemperatureToFahrenheit()
            2 -> convertTemperatureToCelsius()
            3 -> convertKilometersToMiles()
            4 -> convertMilesToKilometers()
            5 -> convertKilogramsToPounds()
            6 -> convertPoundsToKilograms()
            0 -> exitProcess(0)
            else -> println("Неверный ввод, попробуйте снова.\n")
        }
    }
}
```

Результат работы программы представлен на рисунке 12.2

```
Конвертер единиц:
1. Температура: Цельсий → Фаренгейт
2. Температура: Фаренгейт → Цельсий
3. Расстояние: Километры → Мили
4. Расстояние: Мили → Километры
5. Вес: Килограммы → Фунты
6. Вес: Фунты → Килограммы
0. Выход
Выберите опцию: 1
Введите температуру в °C: 45
45.0 °C = 113.0 °F

Конвертер единиц:
1. Температура: Цельсий → Фаренгейт
2. Температура: Фаренгейт → Цельсий
3. Расстояние: Километры → Мили
4. Расстояние: Мили → Километры
5. Вес: Килограммы → Фунты
6. Вес: Фунты → Килограммы
0. Выход
Выберите опцию: 3
Введите расстояние в км: 76
76.0 км = 47.224196 миль
```

Рисунок 12.2 – Результат выполнения программы

3. Вывод

Были изучены nullable-типы, безопасные вызовы и функции-расширения. Реализованы системы профилей пользователей и конвертер единиц измерения, что позволило освоить безопасную работу с данными и расширяемость кода в Kotlin.

ПРАКТИЧЕСКАЯ РАБОТА 13

1. Цель работы

Освоить работу с коллекциями в Kotlin: списками, множествами, ассоциативными массивами, а также методы их обработки (фильтрация, группировка, сортировка).

2. Решение

2.1.Задание 1

Создайте приложение для анализа заказов в интернет-магазине. Каждый заказ представлен объектом с полями orderId, customerId, products (список названий продуктов), totalPrice (сумма заказа).

Создайте изменяемую коллекцию из нескольких заказов.

Реализуйте функции для:

1. Добавления нового заказа в коллекцию.
2. Удаления заказов, сумма которых меньше заданного значения.
3. Получения списка всех уникальных клиентов (по customerId).
4. Подсчета общего дохода магазина по всем заказам.
5. Получения топ-3 самых дорогих заказов.

Был создан менеджер заказов OrderManager, работающий с изменяемой коллекцией заказов. Метод addOrder() добавлял новый заказ в список. Метод removeOrdersBelowPrice() с помощью removeAll и лямбда-выражения удалял все заказы, сумма которых была ниже заданного порога. Метод getUniqueCustomers() с использованием map и toSet() возвращал множество уникальных идентификаторов клиентов. Метод getTotalRevenue() с помощью sumOf вычислял общую сумму всех заказов. Метод getTop3MostExpensiveOrders() комбинировал sortedByDescending для сортировки по убыванию суммы и take(3) для получения трёх самых дорогих заказов.

Код реализации алгоритма программы представлен в листингах 13.1 – 13.3

Листинг 13.1 – Data-класс Order

```
package pract_13.task_1

data class Order(
    val orderId: Int,
    val customerId: Int,
    val products: List<String>,
    val totalPrice: Double
)
```

Листинг 13.2 – Класс OrderManager

```
package pract_13.task_1

class OrderManager {
    private val orders = mutableListOf<Order>()

    fun addOrder(order: Order) {
        orders.add(order)
    }

    fun removeOrdersBelowPrice(threshold: Double) {
        orders.removeAll { it.totalPrice < threshold }
    }

    fun getUniqueCustomers(): Set<Int> {
        return orders.map { it.customerId }.toSet()
    }

    fun getTotalRevenue(): Double {
        return orders.sumOf { it.totalPrice }
    }

    fun getTop3MostExpensiveOrders(): List<Order> {
        return orders.sortedByDescending { it.totalPrice }.take(3)
    }

    fun displayAllOrders() {
        println("\nВсе заказы:")
        orders.forEach { order ->
            println("Заказ №${order.orderId}, Клиент: ${order.customerId}, "
+
                    "Товары: ${order.products}, Сумма: ${order.totalPrice}")
        }
    }
}
```

Листинг 13.3 – Функция main

```
package pract_13.task_1

fun main() {
    val orderManager = OrderManager()

    val order1 = Order(1, 1, listOf("Ноутбук", "Мышь"), 75000.0)
    val order2 = Order(2, 2, listOf("Смартфон"), 45000.0)
    val order3 = Order(3, 1, listOf("Наушники", "Чехол"), 12000.0)
    val order4 = Order(4, 3, listOf("Планшет", "Клавиатура"), 55000.0)
    val order5 = Order(5, 4, listOf("Монитор"), 30000.0)

    orderManager.addOrder(order1)
    orderManager.addOrder(order2)
    orderManager.addOrder(order3)
    orderManager.addOrder(order4)
    orderManager.addOrder(order5)
```

```
    orderManager.displayAllOrders()

    println("\nУникальные клиенты: ${orderManager.getUniqueCustomers()}")

    println("\nОбщая выручка: ${orderManager.getTotalRevenue()} руб.")

    println("\nТоп 3 самых дорогих заказа:")
    orderManager.getTop3MostExpensiveOrders().forEachIndexed { index, order
->
        println("${index + 1}. Заказ №${order.orderId} - ${order.totalPrice}
руб.")
    }

    println("\nУдаление заказов до 40000 руб:")
    orderManager.removeOrdersBelowPrice(40000.0)

    orderManager.displayAllOrders()
}
```

Результат работы программы представлен на рисунке 13.1

```
Все заказы:
Заказ №1, Клиент: 1, Товары: [Ноутбук, Мышь], Сумма: 75000.0
Заказ №2, Клиент: 2, Товары: [Смартфон], Сумма: 45000.0
Заказ №3, Клиент: 1, Товары: [Наушники, Чехол], Сумма: 12000.0
Заказ №4, Клиент: 3, Товары: [Планшет, Клавиатура], Сумма: 55000.0
Заказ №5, Клиент: 4, Товары: [Монитор], Сумма: 30000.0

Уникальные клиенты: [1, 2, 3, 4]

Общая выручка: 217000.0 руб.

Топ 3 самых дорогих заказа:
1. Заказ №1 - 75000.0 руб.
2. Заказ №4 - 55000.0 руб.
3. Заказ №2 - 45000.0 руб.

Удаление заказов до 40000 руб:

Все заказы:
Заказ №1, Клиент: 1, Товары: [Ноутбук, Мышь], Сумма: 75000.0
Заказ №2, Клиент: 2, Товары: [Смартфон], Сумма: 45000.0
Заказ №4, Клиент: 3, Товары: [Планшет, Клавиатура], Сумма: 55000.0
```

Рисунок 13.1 – Результат выполнения программы

2.2.Задание 2

Создайте программу для анализа данных о продажах. Каждая продажа представлена объектом с полями date (дата продажи), product, quantity, pricePerUnit.

Создайте неизменяемую коллекцию из нескольких продаж.

Реализуйте:

1. Подсчет общей выручки за все время.
2. Группировку продаж по продуктам и подсчет количества проданных единиц каждого продукта.
3. Получение списка продаж за конкретный месяц.
4. Определение продукта, который принес наибольшую выручку.

Был разработан анализатор продаж SalesAnalyzer, работающий с неизменяемой коллекцией. Data-класс Sale содержал вычисляемое свойство totalPrice, которое автоматически рассчитывало общую стоимость продажи. Метод getSalesByProduct() использовал groupBy для группировки продаж по продуктам и mapValues для преобразования каждой группы в общее количество проданных единиц. Метод getSalesByMonth() с помощью filter отбирал продажи за указанный месяц и год. Метод getTopProductByRevenue() сначала группировал продажи по продуктам, затем вычислял общую выручку по каждому продукту и с помощью maxByOrNull находил продукт с максимальной выручкой.

Код реализации алгоритма программы представлен в листингах 13.4 – 13.6

Листинг 13.4 – Data-класс Sale

```
package pract_13.task_2

import java.time.LocalDate

data class Sale(
    val date: LocalDate,
    val product: String,
    val quantity: Int,
    val pricePerUnit: Double
) {
    val totalPrice: Double
        get() = quantity * pricePerUnit
}
```

Листинг 13.5 – Класс SalesAnalyzer

```
package pract_13.task_2

class SalesAnalyzer(private val sales: List<Sale>) {

    fun getTotalRevenue(): Double {
        return sales.sumOf { it.totalPrice }
    }

    fun getSalesByProduct(): Map<String, Int> {
        return sales.groupBy { it.product } .mapValues { (_, sales) ->
sales.sumOf { it.quantity } }
    }

    fun getSalesByMonth(month: Int, year: Int): List<Sale> {
        return sales.filter {
            it.date.monthValue == month && it.date.year == year
        }
    }

    fun getTopProductByRevenue(): String {
        return sales.groupBy { it.product }
            .mapValues { (_, sales) -> sales.sumOf { it.totalPrice } }
            .maxByOrNull { it.value }
            ?.key ?: "Нет данных"
    }

    fun displayAllSales() {
        println("\nВсе продажи:")
        sales.forEach { sale ->
            println("${sale.date}: ${sale.product} x${sale.quantity} " +
                    "по ${sale.pricePerUnit} руб. = ${sale.totalPrice}
руб.")
        }
    }
}
```

Листинг 13.6 – Функция main

```
package pract_13.task_2

import java.time.LocalDate

fun main() {
    val sales = listOf(
        Sale(LocalDate.of(2025, 1, 15), "Ноутбук", 2, 50000.0),
        Sale(LocalDate.of(2025, 1, 20), "Смартфон", 5, 30000.0),
        Sale(LocalDate.of(2025, 2, 10), "Ноутбук", 1, 55000.0),
        Sale(LocalDate.of(2025, 2, 15), "Наушники", 10, 5000.0),
        Sale(LocalDate.of(2025, 2, 25), "Смартфон", 3, 32000.0),
        Sale(LocalDate.of(2025, 3, 5), "Планшет", 4, 25000.0)
    )

    val analyzer = SalesAnalyzer(sales)

    analyzer.displayAllSales()

    println("\nОбщая выручка: ${analyzer.getTotalRevenue()} руб.")

    println("\nПродажи по продуктам:")
    analyzer.getSalesByProduct().forEach { (product, quantity) ->
        println("$product: $quantity шт.")
    }
```

```
    println("\nПродажи за февраль 2025:")
    analyzer.getSalesByMonth(2, 2025).forEach { sale ->
        println("${sale.date}: ${sale.product} - ${sale.totalPrice} руб.")
    }

    println("\nПродукт с наибольшей выручкой:
${analyzer.getTopProductByRevenue()}")
}
```

Результат работы программы представлен на рисунке 13.2

```
Все продажи:
2025-01-15: Ноутбук x2 по 50000.0 руб. = 100000.0 руб.
2025-01-20: Смартфон x5 по 30000.0 руб. = 150000.0 руб.
2025-02-10: Ноутбук x1 по 55000.0 руб. = 55000.0 руб.
2025-02-15: Наушники x10 по 5000.0 руб. = 50000.0 руб.
2025-02-25: Смартфон x3 по 32000.0 руб. = 96000.0 руб.
2025-03-05: Планшет x4 по 25000.0 руб. = 100000.0 руб.

Общая выручка: 551000.0 руб.

Продажи по продуктам:
Ноутбук: 3 шт.
Смартфон: 8 шт.
Наушники: 10 шт.
Планшет: 4 шт.

Продажи за февраль 2025:
2025-02-10: Ноутбук - 55000.0 руб.
2025-02-15: Наушники - 50000.0 руб.
2025-02-25: Смартфон - 96000.0 руб.

Продукт с наибольшей выручкой: Смартфон
```

Рисунок 13.2 – Результат выполнения программы

3. Вывод

В ходе работы изучены методы работы с коллекциями: фильтрация, группировка, сортировка, агрегация. Реализованы системы анализа заказов и продаж, что позволило освоить эффективную обработку данных в Kotlin.

ПРАКТИЧЕСКАЯ РАБОТА 14

1. Цель работы

Освоить операции с множествами, работу с ассоциативными списками, фильтрацию, группировку и статистический анализ данных.

2. Решение

2.1.Задание 1

У вас есть два списка чисел:

- listA - набор чисел, представляющий ID товаров, которые покупались за последний месяц.
- listB - набор чисел, представляющий ID товаров, которые находятся на складе.

Найдите:

1. Товары, которые покупались и находятся на складе.
2. Товары, которые покупались, но отсутствуют на складе.
3. Объедините все товары в общий список уникальных ID.

Подсказка: необходимо использовать операции со множествами.

Были продемонстрированы операции над множествами. Списки товаров преобразовывались в множества неявно при вызове методов. Метод intersect() находил пересечение двух множеств — товары, которые и покупались, и есть на складе. Метод subtract() находил разность множеств — товары, которые покупались, но отсутствуют на складе. Метод union() объединял оба множества, удаляя дубликаты, и возвращал все уникальные товары. Эти операции были выполнены в одну строку каждая, демонстрируя выразительность Kotlin при работе с множествами.

Код реализации алгоритма программы представлен в листинге 14.1

Листинг 14.1 – Код программы

```
package pract_14

fun main() {
    val listA = listOf(1, 2, 5, 7, 8, 10, 12, 15, 20)
    val listB = listOf(2, 3, 5, 6, 8, 9, 11, 15, 18)

    val intersection = listA.intersect(listB)
    println("1. Товары, которые покупались и есть на складе: $intersection")
```

```
    val onlyInA = listA.subtract(listB)
    println("2. Товары, которые покупались, но отсутствуют на складе:
$onlyInA")

    val union = listA.union(listB)
    println("3. Все уникальные товары (объединение): $union")
}
```

Результат работы программы представлен на рисунке 14.1

```
1. Товары, которые покупались и есть на складе: [2, 5, 8, 15]
2. Товары, которые покупались, но отсутствуют на складе: [1, 7, 10, 12, 20]
3. Все уникальные товары (объединение): [1, 2, 5, 7, 8, 10, 12, 15, 20, 3, 6, 9, 11, 18]
```

Рисунок 14.1 – Результат выполнения программы

2.2.Задание 2

Вы создаете приложение для коллекционирования и категоризации любимых мемов/цитат из фильмов или строчек из стихотворений.

Исходные данные:

Создайте список entries, где каждый элемент - это ассоциативный список (Map) со следующими ключами:

- «text» – строка, содержащая сам текст (например, цитату, строчку из стихотворения или подпись к мему).
- «author» – строка, содержащая автора (например, «Шекспир», «А.С. Пушкин», «Персонаж Игры Престолов», «Internet Meme»).
- «type» – строка, категория («quote», «poem», «meme»).
- «tags» – список строк, ключевые слова, описывающие запись (например, [«мотивация», «юмор», «жизнь»]).

Необходимо реализовать:

1. Фильтрацию по типу

Создайте функцию, которая возвращает все записи определенного типа (например, только цитаты или только мемы).

2. Поиск по автору

Создайте функцию, которая возвращает все записи указанного автора.

3. Поиск по тегу

Создайте функцию, которая возвращает все записи, содержащие определенный тег (например, все записи с тегом «юмор»).

4. Статистику

Получите множество всех уникальных авторов в вашей библиотеке. Получите множество всех уникальных тегов, которые используются в библиотеке.

5. Группировку

Сгруппируйте все записи по их типу. Результатом должна быть мапа, где ключ - это тип («quote», «poem», «meme»), а значение - список всех записей этого типа.

Для демонстрации создайте консольное меню.

Была разработана система для каталогизации цитат, стихов и мемов. Данные хранились в виде списка ассоциативных массивов (Map<String, Any>), где каждый элемент содержал текст, автора, тип и список тегов. Для фильтрации записей использовался метод filter с соответствующим условием. Поиск по тегам выполнялся с помощью комбинации filter и any, которая проверяла, содержится ли искомый тег в списке тегов записи. Метод getAllUniqueAuthors() с использованием map и toSet() извлекал всех уникальных авторов. Метод getAllUniqueTags() с помощью flatMap "разворачивал" все списки тегов в один плоский список, а затем преобразовывал его в множество для удаления дубликатов. Метод groupByType() использовал groupBy для создания словаря, где ключами были типы записей, а значениями — списки записей соответствующего типа. Вся система была обёрнута в интерактивное консольное меню для удобного тестирования функционала.

Код реализации алгоритма программы представлен в листингах 14.2 – 14.3

Листинг 14.2 – Класс MemeQuoteDatabase

```
package pract_14.task_2

class MemeQuoteDatabase {
    private val entries = mutableListOf<Map<String, Any>>()

    fun addNewEntry(newEntry : Map<String, Any>) {
        entries.add(newEntry)
    }
```

```
    fun filterByType(type: String): List<Map<String, Any>> {
        return entries.filter { it["type"] == type }
    }

    fun findByAuthor(author: String): List<Map<String, Any>> {
        return entries.filter { (it["author"] as String).equals(author,
ignoreCase = true) }
    }

    fun findByTag(tag: String): List<Map<String, Any>> {
        return entries.filter { entry ->
            val tags = entry["tags"] as List<String>
            tags.any { it.equals(tag, ignoreCase = true) }
        }
    }

    fun getAllUniqueAuthors(): Set<String> {
        return entries.map { it["author"] as String }.toSet()
    }

    fun getAllUniqueTags(): Set<String> {
        return entries.flatMap { it["tags"] as List<String> }.toSet()
    }

    fun groupByType(): Map<String, List<Map<String, Any>>> {
        return entries.groupBy { it["type"] as String }
    }

    fun printEntry(entry: Map<String, Any>, index: Int = -1) {
        val prefix = if (index > 0) "$index. " else "   "
        println("${prefix}Текст: ${entry["text"]}")
        println("${" ".repeat(prefix.length)}Автор: ${entry["author"]}")
        println("${" ".repeat(prefix.length)}Тип: ${entry["type"]}")
        println("${" ".repeat(prefix.length)}Теги: ${(entry["tags"] as
List<String>).joinToString(", ")}\n")
    }

    fun printAllEntries() {
        println("Все записи в БД:")
        entries.forEachIndexed { i, entry ->
            println("\nЗапись ${i + 1}:")
            printEntry(entry)
        }
    }
}
```

Листинг 14.3 – Функция main

```
package pract_14.task_2

import kotlin.collections.chunked
import kotlin.collections.joinToString
import kotlin.system.exitProcess

fun addTestData(database: MemeQuoteDatabase) {
    database.addNewEntry(mapOf(
        "text" to "Быть или не быть, вот в чем вопрос",
        "author" to "Шекспир",
        "type" to "quote",
        "tags" to listOf("философия", "жизнь", "решение")
    ))

    database.addNewEntry(mapOf(
        "text" to "Я помню чудное мгновенье",
```

```
        "author" to "А.С. Пушкин",
        "type" to "poem",
        "tags" to listOf("любовь", "воспоминание", "красота")
    ))

    database.addNewEntry(mapOf(
        "text" to "Это просто бизнес, ничего личного",
        "author" to "Крестный отец",
        "type" to "quote",
        "tags" to listOf("бизнес", "кино", "серьезность")
    ))

    database.addNewEntry(mapOf(
        "text" to "Я к вам пишу — чего же боле?",
        "author" to "А.С. Пушкин",
        "type" to "poem",
        "tags" to listOf("любовь", "письмо", "эмоции")
    ))

    database.addNewEntry(mapOf(
        "text" to "Это СПАРТА!!!",
        "author" to "300 спартанцев",
        "type" to "meme",
        "tags" to listOf("кино", "мотивация", "эпично")
    ))

    database.addNewEntry(mapOf(
        "text" to "Даже не знаю, что сказать",
        "author" to "Internet Meme",
        "type" to "meme",
        "tags" to listOf("юмор", "растерянность", "интернет")
    ))
}

fun filterByType(database: MemeQuoteDatabase) {
    print("\nВведите тип для фильтрации: ")
    val type = readLine().toString()
    println("\nФильтрация по типу '${type}':")
    val search = database.filterByType(type)
    if (search.isEmpty()) {
        println("  Записей не найдено")
    } else {
        search.forEachIndexed { i, entry -> database.printEntry(entry, i +
1) }
    }
}

fun findByAuthor(database: MemeQuoteDatabase) {
    print("\nВведите искомого автора: ")
    val author = readLine().toString()
    println("\nПоиск по автору '${author}':")
    val search = database.findByAuthor(author)
    if (search.isEmpty()) {
        println("  Записей не найдено")
    } else {
        search.forEachIndexed { i, entry -> database.printEntry(entry, i +
1) }
    }
}

fun findByTag(database: MemeQuoteDatabase) {
    print("\nВведите искомый тег: ")
    val tag = readLine().toString()
    println("\nПоиск по тегу '${tag}':")
```

```
    val search = database.findByTag(tag)
    if (search.isEmpty()) {
        println("  Записей не найдено")
    } else {
        search.forEachIndexed { i, entry -> database.printEntry(entry, i +
1) }
    }
}

fun showStatistic(database: MemeQuoteDatabase) {
    println("\nСтатистика:")
    val uniqueAuthors = database.getAllUniqueAuthors()
    val uniqueTags = database.getAllUniqueTags()

    println("  Уникальные авторы (${uniqueAuthors.size}):")
    uniqueAuthors.forEachIndexed { i, author ->
        println("    ${i + 1}. $author")
    }

    println("\n  Уникальные теги (${uniqueTags.size}):")
    uniqueTags.chunked(4).forEachIndexed { i, chunk ->
        println("    ${chunk.joinToString(", ")}")
    }
    println()
}

fun groupByType(database: MemeQuoteDatabase) {
    println("\nГруппировка по типу:")
    val groupedByType = database.groupByType()

    groupedByType.forEach { (type, typeEntries) ->
        println("\n  Тип: '$type' (${typeEntries.size} записей)")
        typeEntries.forEachIndexed { i, entry ->
            println("    ${i + 1}. ${entry["text"]}")
            println("       Автор: ${entry["author"]}")
        }
    }
    println()
}

fun main() {
    val database = MemeQuoteDatabase()
    addTestData(database)

    while (true) {
        println("1. Фильтр по типу")
        println("2. Поиск по автору")
        println("3. Поиск по тегу")
        println("4. Показать статистику")
        println("5. Сгруппировать по типам")
        println("6. Вывести всю БД")
        println("0. Выход")
        print("Выберите опцию: ")

        when (readLine()?.toIntOrNull()) {
            1 -> filterByType(database)
            2 -> findByAuthor(database)
            3 -> findByTag(database)
            4 -> showStatistic(database)
            5 -> groupByType(database)
            6 -> database.printAllEntries()
            0 -> exitProcess(0)
            else -> println("Неверный ввод, попробуйте снова.\n")
        }
```

```
    }
}
```

Результат работы программы представлен на рисунках 14.2 – 14.6

```
1. Фильтр по типу
2. Поиск по автору
3. Поиск по тегу
4. Показать статистику
5. Сгруппировать по типам
6. Вывести всю БД
0. Выход
Выберите опцию: 1

Введите тип для фильтрации: meme

Фильтрация по типу 'meme':
1. Текст: Это СПАРТА!!!
   Автор: 300 спартанцев
   Тип: meme
   Теги: кино, мотивация, эпично

2. Текст: Даже не знаю, что сказать
   Автор: Internet Meme
   Тип: meme
   Теги: юмор, растерянность, интернет
```

Рисунок 14.2 – Результат выполнения программы (часть 1)

```
1. Фильтр по типу
2. Поиск по автору
3. Поиск по тегу
4. Показать статистику
5. Сгруппировать по типам
6. Вывести всю БД
0. Выход
Выберите опцию: 2

Введите искомого автора: А.С. Пушкин

Поиск по автору 'А.С. Пушкин':
1. Текст: Я помню чудное мгновенье
   Автор: А.С. Пушкин
   Тип: poem
   Теги: любовь, воспоминание, красота

2. Текст: Я к вам пишу — чего же боле?
   Автор: А.С. Пушкин
   Тип: poem
   Теги: любовь, письмо, эмоции
```

Рисунок 14.3 – Результат выполнения программы (часть 2)

```
1. Фильтр по типу
2. Поиск по автору
3. Поиск по тегу
4. Показать статистику
5. Сгруппировать по типам
6. Вывести всю БД
0. Выход
Выберите опцию: 3

Введите искомый тег: кино

Поиск по тегу 'кино':
1. Текст: Это просто бизнес, ничего личного
   Автор: Крестный отец
   Тип: quote
   Теги: бизнес, кино, серьезность

2. Текст: Это СПАРТА!!!
   Автор: 300 спартанцев
   Тип: meme
   Теги: кино, мотивация, эпично
```

Рисунок 14.4 – Результат выполнения программы (часть 3)

```
1. Фильтр по типу
2. Поиск по автору
3. Поиск по тегу
4. Показать статистику
5. Сгруппировать по типам
6. Вывести всю БД
0. Выход
Выберите опцию: 4

Статистика:
  Уникальные авторы (5):
    1. Шекспир
    2. А.С. Пушкин
    3. Крестный отец
    4. 300 спартанцев
    5. Internet Meme

  Уникальные теги (16):
    философия, жизнь, решение, любовь
    воспоминание, красота, бизнес, кино
    серьезность, письмо, эмоции, мотивация
    эпично, юмор, растерянность, интернет
```

Рисунок 14.5 – Результат выполнения программы (часть 4)

```
1. Фильтр по типу
2. Поиск по автору
3. Поиск по тегу
4. Показать статистику
5. Сгруппировать по типам
6. Вывести всю БД
0. Выход
Выберите опцию: 5

Группировка по типу:

  Тип: 'quote' (2 записей)
    1. Быть или не быть, вот в чем вопрос
       Автор: Шекспир
    2. Это просто бизнес, ничего личного
       Автор: Крестный отец

  Тип: 'poem' (2 записей)
    1. Я помню чудное мгновенье
       Автор: А.С. Пушкин
    2. Я к вам пишу — чего же боле?
       Автор: А.С. Пушкин

  Тип: 'meme' (2 записей)
    1. Это СПАРТА!!!
       Автор: 300 спартанцев
    2. Даже не знаю, что сказать
       Автор: Internet Meme
```

Рисунок 14.6 – Результат выполнения программы (часть 5)

3. Вывод

Были изучены операции с множествами, работа с ассоциативными списками, фильтрация и группировка данных. Реализованы системы анализа товаров и базы цитат, что позволило освоить методы обработки сложных структур данных в Kotlin.

ПРАКТИЧЕСКАЯ РАБОТА 15

1. **Цель работы**

Освоить работу с последовательностями (sequences) в Kotlin, их методы обработки данных (фильтрация, трансформация, группировка, сортировка), а также научиться выполнять статистический анализ данных с использованием встроенных функций Kotlin.

2. **Решение**

Создайте программу для анализа данных о загрязнении воздуха в городах.

Данные:

```
data class CityAirQuality(
    val city: String,
    val country: String,
    val pm25: Int, // концентрация PM2.5
    val pm10: Int,
    val lastUpdate: String
)
```

Необходимо релизовать:

1. Создание последовательности из 10–15 городов с разными показателями качества воздуха.
2. Фильтрацию по уровню – выбрать города, где PM2.5 > 50 (опасный уровень).
3. Проверку безопасности – проверить, все ли города в определенной стране имеют PM10 < 100.
4. Трансформацию в статистику – преобразовать данные в строку: «Город: [city], PM2.5: [pm25], Статус: ${if(pm25 > 50) "Опасно" else "Нормально"}»
5. Группировку города по странам.
6. Сортировку города по убыванию PM2.5.
7. Расчет для каждой страны средний PM2.5 и PM10.

Для создания последовательности использовалась функция sequenceOf(). Для фильтрации данных применялся метод filter() с условием it.pm25 > 50. Для проверки безопасности в стране использована комбинация filter() и all() с условием it.pm10 < 100. Трансформация данных в строку выполнялась через map() с использованием условного оператора if для определения статуса. Группировка по странам осуществлялась методом groupBy(). Сортировка по убыванию PM2.5 выполнялась через sortedByDescending(). Для расчёта средних значений использовались map(), average() и форматирование через "%.2f".format(). Также применялись циклы forEach() для вывода результатов.

Код реализации алгоритма программы представлен в листингах 15.1 – 15.2

Листинг 15.1 – Data-класс CityAirQuality

```
package pract_15

data class CityAirQuality(
    val city: String,
    val country: String,
    val pm25: Int,
    val pm10: Int,
    val lastUpdate: String
)
```

Листинг 15.2 – Функция main

```
package pract_15

fun main() {
    val cities = sequenceOf(
        CityAirQuality("Москва", "Россия", 65, 120, "2025-03-15"),
        CityAirQuality("Санкт-Петербург", "Россия", 40, 90, "2025-03-15"),
        CityAirQuality("Новосибирск", "Россия", 70, 130, "2025-03-15"),
        CityAirQuality("Лондон", "Великобритания", 30, 60, "2025-03-14"),
        CityAirQuality("Манчестер", "Великобритания", 55, 110, "2025-03-
14"),
        CityAirQuality("Пекин", "Китай", 120, 200, "2025-03-13"),
        CityAirQuality("Шанхай", "Китай", 110, 190, "2025-03-13"),
        CityAirQuality("Берлин", "Германия", 25, 50, "2025-03-15"),
        CityAirQuality("Гамбург", "Германия", 35, 70, "2025-03-15"),
        CityAirQuality("Париж", "Франция", 45, 85, "2025-03-15"),
        CityAirQuality("Лион", "Франция", 60, 95, "2025-03-15")
    )


    println("1. Список всех городов:")
    cities.forEach { println(it) }
    println()


    val dangerousCities = cities.filter { it.pm25 > 50 }
    println("2. Города с опасным уровнем PM2.5 (>50):")
    dangerousCities.forEach { println("${it.city} (${it.country}): PM2.5 =
${it.pm25}") }
```

```
    println()


    val countryToCheck = "Россия"
    val allSafeInCountry = cities
        .filter { it.country == countryToCheck }
        .all { it.pm10 < 100 }
    println("3. Все ли города в $countryToCheck имеют PM10 < 100?
$allSafeInCountry")
    println()


    println("4. Статистика по городам:")
    cities.map { city ->
        "Город: ${city.city}, PM2.5: ${city.pm25}, Статус: ${if (city.pm25 >
50) "Опасно" else "Нормально"}"
    }.forEach { println(it) }
    println()


    val groupedByCountry = cities.groupBy { it.country }
    println("5. Группировка по странам:")
    groupedByCountry.forEach { (country, list) ->
        println("$country: ${list.map { it.city }}")
    }
    println()


    val sortedByPm25 = cities.sortedByDescending { it.pm25 }
    println("6. Города, отсортированные по убыванию PM2.5:")
    sortedByPm25.forEach { println("${it.city}: PM2.5 = ${it.pm25}") }
    println()


    println("7. Средние показатели по странам:")
    groupedByCountry.forEach { (country, list) ->
        val avgPm25 = list.map { it.pm25 }.average()
        val avgPm10 = list.map { it.pm10 }.average()
        println("$country: средний PM2.5 = ${"%.2f".format(avgPm25)},
средний PM10 = ${"%.2f".format(avgPm10)}")
    }
}
```

Результат работы программы представлен на рисунках 15.1 – 15.2

```
2. Города с опасным уровнем PM2.5 (>50):
Москва (Россия): PM2.5 = 65
Новосибирск (Россия): PM2.5 = 70
Манчестер (Великобритания): PM2.5 = 55
Пекин (Китай): PM2.5 = 120
Шанхай (Китай): PM2.5 = 110
Лион (Франция): PM2.5 = 60

3. Все ли города в Россия имеют PM10 < 100? false

4. Статистика по городам:
Город: Москва, PM2.5: 65, Статус: Опасно
Город: Санкт-Петербург, PM2.5: 40, Статус: Нормально
Город: Новосибирск, PM2.5: 70, Статус: Опасно
Город: Лондон, PM2.5: 30, Статус: Нормально
Город: Манчестер, PM2.5: 55, Статус: Опасно
Город: Пекин, PM2.5: 120, Статус: Опасно
Город: Шанхай, PM2.5: 110, Статус: Опасно
Город: Берлин, PM2.5: 25, Статус: Нормально
Город: Гамбург, PM2.5: 35, Статус: Нормально
Город: Париж, PM2.5: 45, Статус: Нормально
Город: Лион, PM2.5: 60, Статус: Опасно
```

Рисунок 15.1 – Результат выполнения программы

```
5. Группировка по странам:
Россия: [Москва, Санкт-Петербург, Новосибирск]
Великобритания: [Лондон, Манчестер]
Китай: [Пекин, Шанхай]
Германия: [Берлин, Гамбург]
Франция: [Париж, Лион]

6. Города, отсортированные по убыванию PM2.5:
Пекин: PM2.5 = 120
Шанхай: PM2.5 = 110
Новосибирск: PM2.5 = 70
Москва: PM2.5 = 65
Лион: PM2.5 = 60
Манчестер: PM2.5 = 55
Париж: PM2.5 = 45
Санкт-Петербург: PM2.5 = 40
Гамбург: PM2.5 = 35
Лондон: PM2.5 = 30
Берлин: PM2.5 = 25

7. Средние показатели по странам:
Россия: средний PM2.5 = 58,33, средний PM10 = 113,33
Великобритания: средний PM2.5 = 42,50, средний PM10 = 85,00
Китай: средний PM2.5 = 115,00, средний PM10 = 195,00
Германия: средний PM2.5 = 30,00, средний PM10 = 60,00
Франция: средний PM2.5 = 52,50, средний PM10 = 90,00
```

Рисунок 15.2 – Результат выполнения программы

3. Вывод

В ходе выполнения работы были изучены методы работы с последовательностями (sequences) в Kotlin, включая фильтрацию, сортировку, группировку и статистическую обработку данных. Реализована программа для анализа качества воздуха в городах, что позволило закрепить навыки применения встроенных функций Kotlin для обработки структур данных и выполнения аналитических операций. Были освоены такие методы, как filter(), map(), groupBy(), sortedByDescending(), average(), а также работа с условными конструкциями и форматированием вывода.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Синтаксис языка Котлин [Электронный ресурс]. - https://kotlinlang.ru/docs/basic-syntax.html (Дата обращения: 10.09.2025).

2. Основные типы данных [Электронный ресурс]. - https://kotlinlang.ru/docs/basic-types.html (Дата обращения: 17.09.2025).

3. Функции в языке Котлин [Электронный ресурс]. - https://kotlinlang.ru/docs/functions.html (Дата обращения: 20.09.2025).

4. Классы и объекты [Электронный ресурс]. - https://kotlinlang.ru/docs/classes.html (Дата обращения: 29.09.2025).

5. Справочник. Ключевые слова и операторы [Электронный ресурс]. - https://kotlinlang.ru/docs/keyword-reference.html (Дата обращения: 17.10.2025).