|  |  |  |
| --- | --- | --- |
|  |  |  |
| 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 Враженко Д. О.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись студента)

**Принял старший преподаватель** Золотухин С. А.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись руководителя)

Работа представлена «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_ 2025 г.

Допущен к работе «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 2025 г.

Москва 2025

Оглавление

[Практическая работа 1 3](#__RefHeading___Toc1389_1090301320)

[Практическая работа 2 6](#__RefHeading___Toc1391_1090301320)

[Практическая работа 3 10](#__RefHeading___Toc1393_1090301320)

[Практическая работа 4 15](#__RefHeading___Toc1395_1090301320)

[Практическая работа 5 20](#__RefHeading___Toc1397_1090301320)

[Практическая работа 6 25](#__RefHeading___Toc1399_1090301320)

[Практическая работа 7 32](#__RefHeading___Toc1401_1090301320)

[Практическая работа 8 37](#__RefHeading___Toc1403_1090301320)

[Практическая работа 9 44](#__RefHeading___Toc1405_1090301320)

[Практическая работа 10 52](#__RefHeading___Toc1407_1090301320)

[Практическая работа 11 58](#__RefHeading___Toc1409_1090301320)

[Практическая работа 12 64](#__RefHeading___Toc1411_1090301320)

[Практическая работа 13 71](#__RefHeading___Toc1413_1090301320)

[Практическая работа 14 75](#__RefHeading___Toc1415_1090301320)

[Практическая работа 15 82](#__RefHeading___Toc1417_1090301320)

[Практическая работа 16 84](#__RefHeading___Toc1419_1090301320)

# Практическая работа 1

## Цель работы

Изучить основные конструкции языка Котлин. Выполнить практические задания.

## Решение

## 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 была использована функция readln() для считывания входной строки. Для подсчета количества каждого нуклеотида использован цикл for, который проходит по каждому символу строки. Внутри цикла применяется условная конструкция when для определения типа нуклеотида и инкрементации соответствующего счетчика. Результат выводится в консоль с помощью строкового шаблона.

Код реализации алгоритма программы представлен на рисунке 1.1

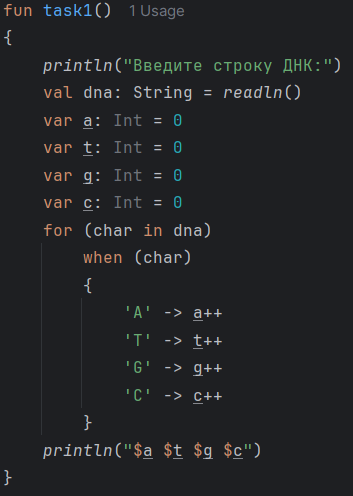


Рисунок 1.1 – Код функции task1()

Результат работы программы представлен на рисунке 1.2

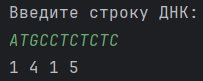


Рисунок 1.2 – Результат выполнения программы для задания 1

## 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).

Задача решается жадным алгоритмом. Для ввода числа использован java.util.Scanner. Алгоритм последовательно находит максимально возможное количество купюр старшего номинала, начиная с 8. Для этого применяются операции целочисленного деления (/) для получения количества купюр и взятия остатка (%) для определения оставшейся суммы после их "выдачи". Результат в виде четырех чисел выводится в консоль.

Код реализации алгоритма программы представлен на рисунке 1.3

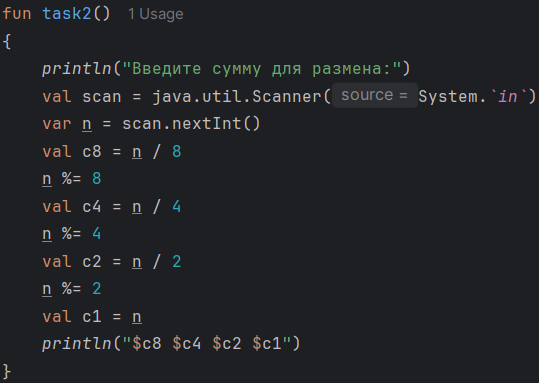


Рисунок 1.3 – Код функции task2()

Результат работы программы представлен на рисунке 1.4

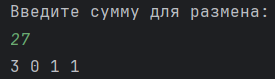


Рисунок 1.4 – Результат выполнения программы для задания 2

## Вывод

В ходе выполнения данной практической работы были изучены базовые конструкции языка Kotlin, такие как объявление функций (fun), переменных (val, var), организация ввода данных с консоли (readln(), Scanner), работа со строками и циклом for, а также использование условного оператора when. Были успешно выполнены задания по обработке строк и реализации простого алгоритма размена денег. Приобретены начальные навыки написания и отладки консольных приложений на Kotlin.

# Практическая работа 2

## Цель работы

Изучить работу с массивами и коллекциями в языке Kotlin. Освоить методы обработки последовательностей данных, включая статистические вычисления, анализ отсортированных массивов и поиск дубликатов.

## Решение

## 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.

Для решения задачи был использован ввод данных через readln() с последующим преобразованием в целое число для определения длины массива. Массив вещественных чисел создан с помощью DoubleArray, где каждый элемент инициализируется преобразованием введенной строки в Double. Для вычисления среднего арифметического использован стандартный метод average(), предоставляемый классом DoubleArray.

Код реализации алгоритма программы представлен на рисунке 2.1

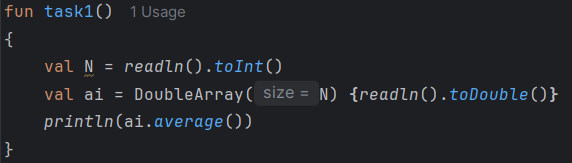


Рисунок 2.1 – Код функции task1()

Результат работы программы представлен на рисунке 2.2

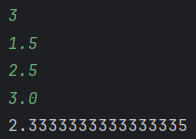


Рисунок 2.2 – Результат выполнения программы для задания 1

## Задание 2

На вход подается число N — длина массива. Затем передается массив целых чисел (*ai*) 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.

Использован алгоритм линейного прохода по отсортированному массиву. С помощью переменных current (текущий элемент) и count (счетчик повторений) осуществляется подсчет одинаковых подряд идущих элементов. При смене элемента выводится накопленная информация о предыдущем элементе. Особое внимание уделено корректному выводу последней группы элементов после завершения цикла.

Код реализации алгоритма программы представлен на рисунке 2.3

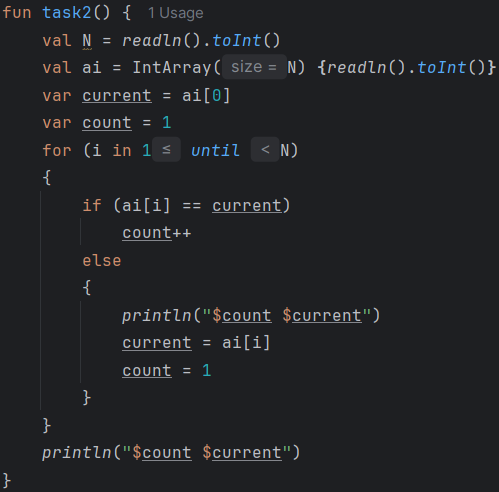


Рисунок 2.3 – Код функции task2()

Результат работы программы представлен на рисунке 2.4



Рисунок 2.4 – Результат выполнения программы для задания 2

## 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.

Для эффективного поиска дубликата использована структура данных MutableSet. При считывании каждой строки проверяется результат операции add(), которая возвращает false, если элемент уже присутствует в множестве. Найденный дубликат сохраняется в переменную p и выводится после обработки всех элементов.

Код реализации алгоритма программы представлен на рисунке 2.5

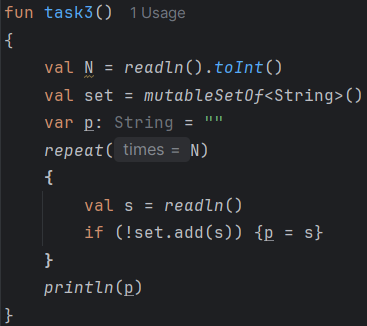


Рисунок 2.5 – Код функции task3()

Результат работы программы представлен на рисунке 2.6

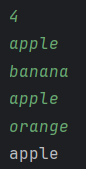


Рисунок 2.6 – Результат выполнения программы для задания 3

## Вывод

В ходе выполнения второй практической работы были изучены основные структуры данных языка Kotlin: массивы (DoubleArray, IntArray). Приобретены навыки работы со стандартными методами массивов, такими как average(). Освоены алгоритмы обработки отсортированных данных и эффективного поиска дубликатов с использованием множеств. Получен практический опыт обработки последовательностей данных различного типа и решения типовых задач анализа информации.

# Практическая работа 3

## Цель работы

Изучить работу с генерацией случайных чисел, обработку строк по сложным правилам, а также реализацию интерактивных консольных приложений с валидацией ввода.

## Решение

## 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.

Алгоритм реализован в отдельной функции guessNumberGame(). Для генерации случайного числа использован Random.nextInt(). Организован бесконечный цикл while(true) с обработкой ввода через readLine()?.toIntOrNull() для корректной обработки ошибок. Логика сравнения реализована с помощью выражения when, которое обрабатывает все возможные случаи: неверный ввод, завершение игры, победу и подсказки.

Код реализации алгоритма программы представлен на рисунке 3.1



Рисунок 3.1 – Код функции guessNumberGame()

Результат работы программы представлен на рисунке 3.2

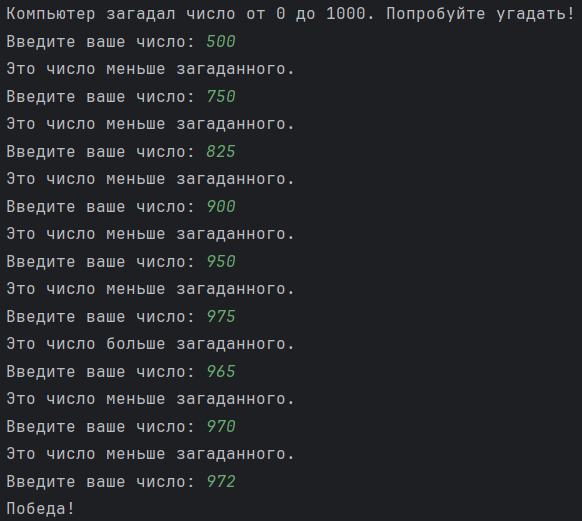


Рисунок 3.2 – Результат выполнения программы для задания 1

## 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:

{".-", "-...", ".--", "--.", "-..", ".", "...-", "--..", "..", ".---", "-.-", ".-..", "--", "-.", "---", ".--.", ".-.", "...", "-", "..-", "..-.", "....", "-.-.", "---.", "----", "--.-", "--.--", "-.--", "-..-", "..-..", "..--", ".-.-"}

Создан массив morseCodes с кодами Морзе для 32 русских букв (А-Я без Ё). Реализована строгая валидация входной строки с использованием регулярного выражения Regex("^[А-Я]+$"). Для кодирования использован цикл по символам строки, где каждый символ преобразуется в индекс массива путем вычитания кода символа 'А'. Результат формируется с помощью StringBuilder для эффективной конкатенации строк.

Код реализации алгоритма программы представлен на рисунке 3.3



Рисунок 3.3 – Код функции encodeToMorse()

Результат работы программы представлен на рисунке 3.4



Рисунок 3.4 – Результат выполнения программы для задания 2

## Дополнительное 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 (\_\*-)

Реализована функция generatePassword() с валидацией длины пароля в цикле. Для обеспечения наличия всех требуемых типов символов сначала добавляется по одному случайному символу из каждой категории, затем оставшаяся длина заполняется символами из объединенного набора. Финальный пароль перемешивается с помощью shuffled() для равномерного распределения символов.

Код реализации алгоритма программы представлен на рисунке 3.5

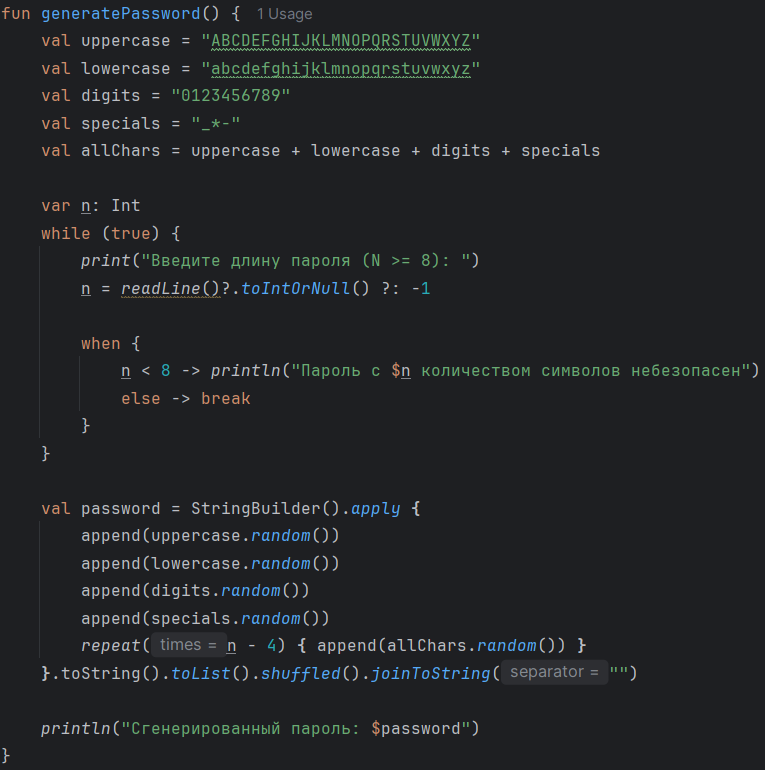


Рисунок 3.5 – Результат написания кода главной функции

Результат работы программы представлен на рисунке 3.6

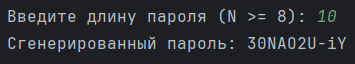


Рисунок 3.6 – Результат работы программы

## Вывод

В ходе выполнения третьей практической работы были успешно освоены принципы модульного программирования в Kotlin. Приобретены навыки работы с генерацией случайных чисел (Random), реализацией интерактивных консольных приложений с цикличным вводом и валидацией данных. Изучены методы обработки строк, включая работу с символами Unicode и использование регулярных выражений для валидации. Особое внимание уделено созданию безопасных алгоритмов генерации паролей, гарантирующих наличие символов всех требуемых категорий.

# Практическая работа 4

## Цель работы

Освоить использование функций различных типов (функции высшего порядка, лямбда-выражения, локальные функции) в языке Kotlin. Закрепить навыки работы с коллекциями и алгоритмами обработки данных. Реализовать консольное меню для интерактивного взаимодействия с пользователем.

## Решение

***При реализации заданий необходимо использовать функции с параметрами, локальные функции и/или лямбда-выражения и/или анонимные функции и/или однострочные функции. Реализовать консольное меню.***

## Задание 1

Вариант 1. Игра «Поле чудес»

Реализуйте консольную версию игры «Поле чудес». Описание:

1. Компьютер загадывает слово (например, из заранее подготовленного списка).
2. На экране отображается строка, в которой на месте букв слова стоят символы \* (звездочка). Например, для слова "ПРОГРАММИРОВАНИЕ" будет показано: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*.
3. Игрок вводит букву.
4. Если буква есть в слове, программа открывает (показывает) все позиции, где эта буква встречается.
5. Если буквы нет, программа сообщает об ошибке и ведет счетчик ошибок.
6. Игра продолжается до тех пор, пока игрок либо не отгадает всё слово, либо не превысит лимит ошибок (например, 10).

Для реализации игры была создана функция task1(), в которой использован разнообразный функционал Kotlin: локальная функция isValidInput() проверяет корректность введенных данных, лямбда-выражение processLetter обрабатывает угадывание букв с обновлением отображаемого слова и подсчетом ошибок, а также применяется случайный выбор слова из подготовленного списка с последующим посимвольным сравнением и отображением прогресса игры через замену символов \* на угаданные буквы.

Код реализации алгоритма программы представлен на рисунках 4.1-4.2

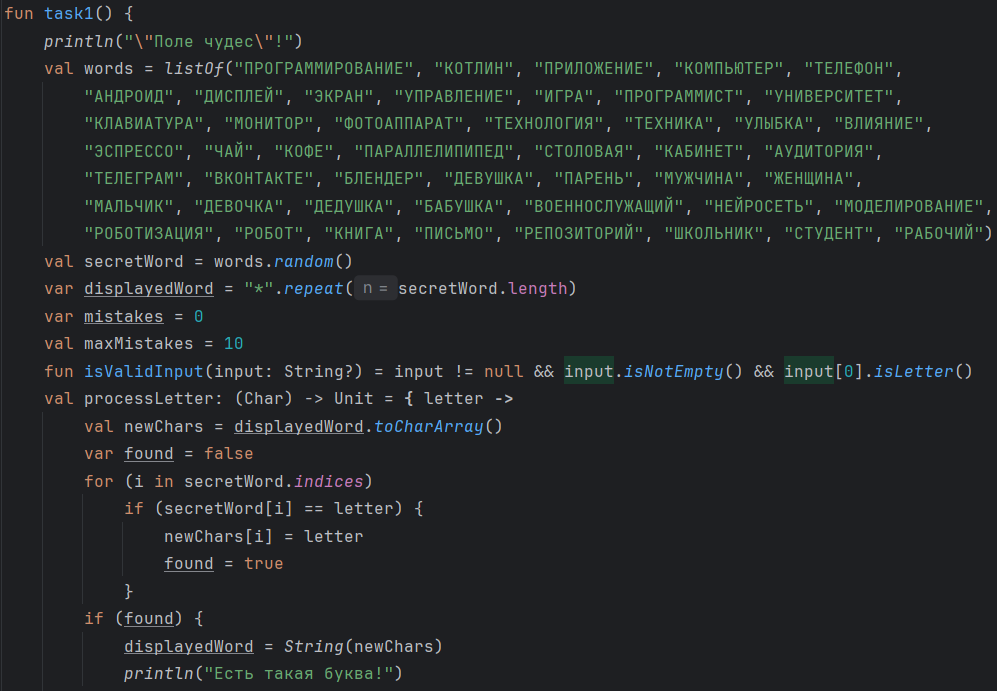


Рисунок 4.1 – Код функции task1() (часть 1)



Рисунок 4.2 – Код функции task1() (часть 2)

Результат работы программы представлен на рисунке 4.3

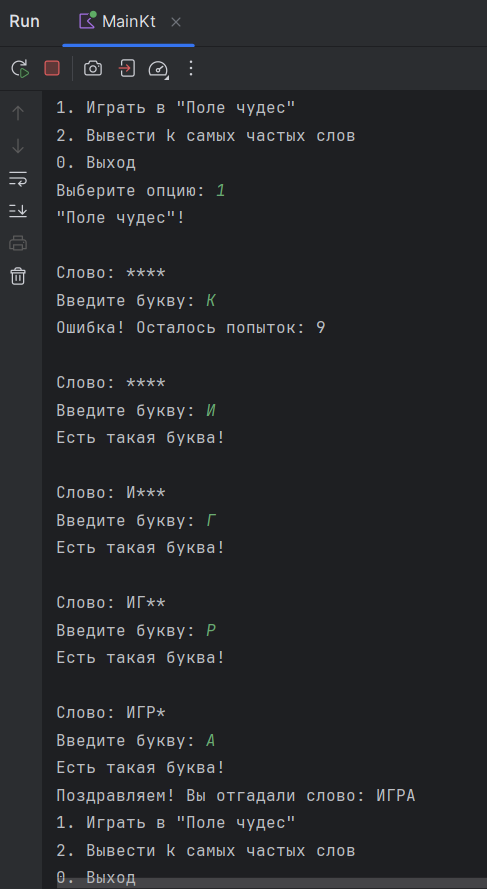


Рисунок 4.3 – Результат выполнения программы для задания 1

## 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.

|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| words = ["the","day","is","sunny","the","the","the", "sunny","is","is","day"] | ["the","is","day","sunny"] |

Функция task2() реализует полный цикл обработки данных: начинается с ввода и валидации исходных данных, затем строится частотный словарь с использованием изменяемой карты и метода getOrDefault(), после чего применяется лямбда-выражение sortByFrequency для реализации алгоритма пузырьковой сортировки с учетом двух критериев - убывания частоты и лексикографического порядка, а завершается процесс формированием и выводом результирующего списка наиболее частых слов.

Код реализации алгоритма программы представлен на рисунках 4.4-4.5

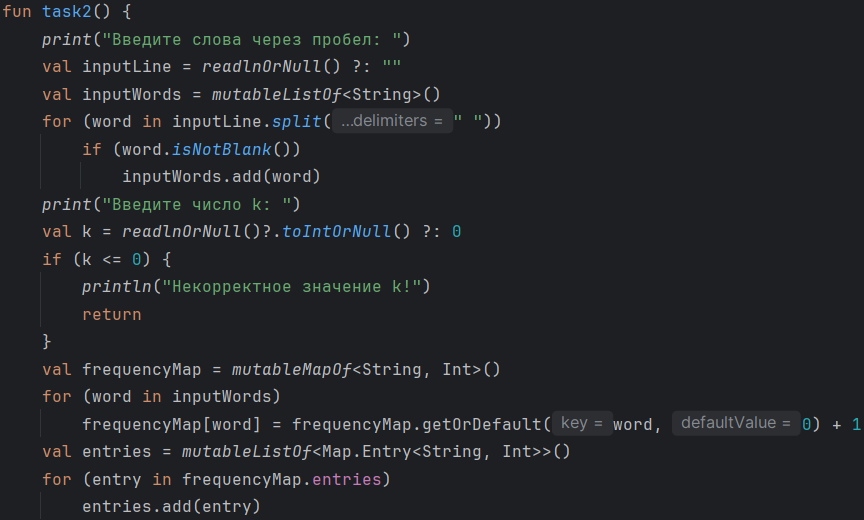


Рисунок 4.4 – Код функции task2() (часть 2)

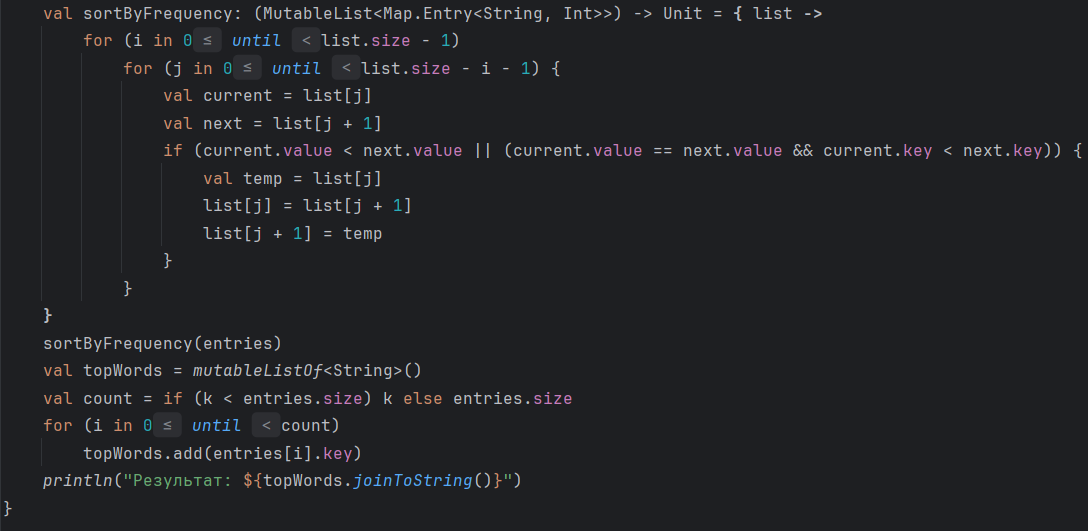


Рисунок 4.5 – Код функции task2() (часть 2)

Результат работы программы представлен на рисунке 4.6

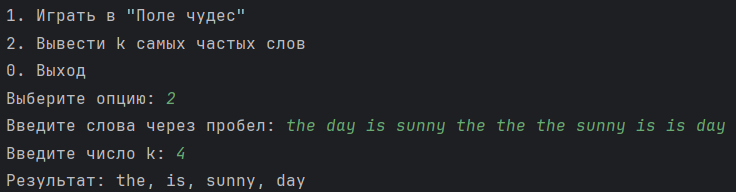


Рисунок 4.6 – Результат работы программы для задания 2

## Вывод

В ходе выполнения четвертой практической работы были успешно освоены различные способы работы с функциями в языке Kotlin. Применены функции высшего порядка, лямбда-выражения и локальные функции для решения практических задач. Реализована сложная интерактивная игра «Поле чудес» с использованием различных типов функций для организации игровой логики. Решена задача статистического анализа текста с сортировкой по нескольким критериям. Приобретены навыки создания консольного меню и организации модульной структуры приложения. Работа продемонстрировала эффективность использования различных типов функций для создания читаемого и поддерживаемого кода.

# Практическая работа 5

## Цель работы

Освоить основы объектно-ориентированного программирования в Kotlin, включая создание классов, свойств с пользовательскими геттерами и сеттерами, методов, а также организацию взаимодействия между классами.

## Решение

## 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, содержащий три приватные функции для различных действий кота, а также публичную функцию randomAction(), которая с помощью генератора случайных чисел и конструкции when выбирает и выполняет одно из приватных действий, демонстрируя принцип инкапсуляции и организацию поведения объекта через внутренние методы.

Код реализации алгоритма программы представлен на рисунке 5.1

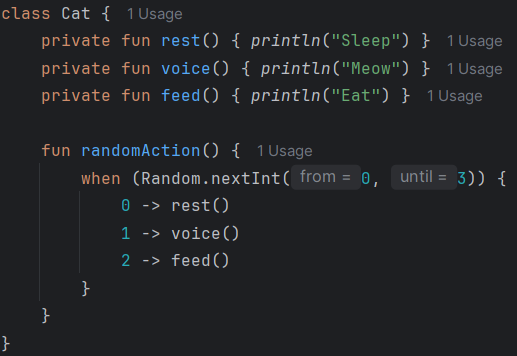


Рисунок 5.1 – Код класса Cat

Результат работы программы представлен на рисунке 5.2

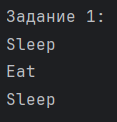


Рисунок 5.2 – Результат выполнения программы для задания 1

## 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**)

Реализован класс Student с приватными свойствами, где для firstName и lastName созданы пользовательские сеттеры, которые обрезают пробелы и приводят первую букву к заглавной, а также методы для управления массивом оценок, включая метод addScore() для добавления новой оценки с сохранением размера массива и метод getAverageScore() для расчета среднего арифметического оценок.

Код реализации алгоритма программы представлен на рисунке 5.3



Рисунок 5.3 – Код класса Student

Результат работы программы представлен на рисунке 5.4

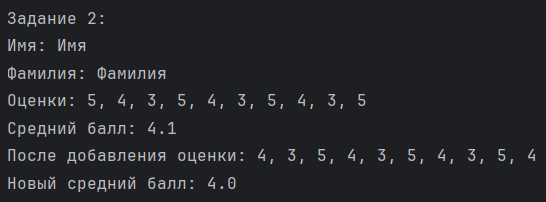


Рисунок 5.4 – Результат выполнения программы для задания 2

## 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.

Создан сервисный класс StudentService, содержащий метод findBestStudent(), который с использованием функции maxByOrNull() находит студента с максимальным средним баллом, и метод sortStudentsByLastName(), который осуществляет сортировку массива студентов по фамилии с помощью sortedBy(), демонстрируя работу с коллекциями объектов и организацию сервисной логики в отдельных классах.

Код реализации алгоритма программы представлен на рисунке 5.5

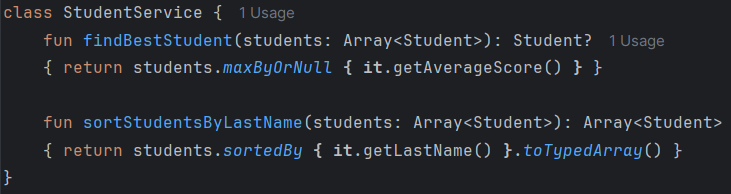


Рисунок 5.5 – Код класса StudentService

Результат работы программы представлен на рисунке 5.6

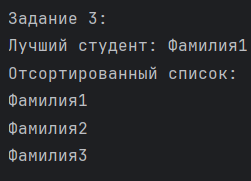


Рисунок 5.6 – Результат выполнения программы для задания 3

## Вывод

В ходе выполнения пятой практической работы были успешно освоены основные принципы объектно-ориентированного программирования в Kotlin, включая создание классов с инкапсулированными свойствами и методами, реализацию пользовательских геттеров и сеттеров для валидации и преобразования данных, организацию взаимодействия между классами через сервисные методы, а также работу с массивами и коллекциями объектов, что позволило создать полноценную систему управления студентами с функционалом расчета статистики и сортировки.

# Практическая работа 6

## Цель работы

Освоить принципы объектно-ориентированного программирования в Kotlin, включая создание классов с несколькими конструкторами, реализацию пользовательских свойств с валидацией, работу с пакетами и организацию сложной бизнес-логики в классах.

## Решение

## 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).

Создан класс TimeMeasure с тремя конструкторами для различных способов инициализации времени, где реализована строгая валидация входных параметров через блок init и приватный метод validateTime(), обеспечивающий корректность значений часов, минут и секунд, а также методы display24HourFormat() и display12HourFormat() для форматированного вывода времени с использованием вспомогательной функции formatTwoDigits() и метод addTime() для выполнения арифметических операций со временем с автоматическим учетом переносов между единицами измерения.

Код реализации алгоритма программы представлен на рисунках 6.1-6.2



Рисунок 6.1 – Код класса TimeMeasure (часть 1)

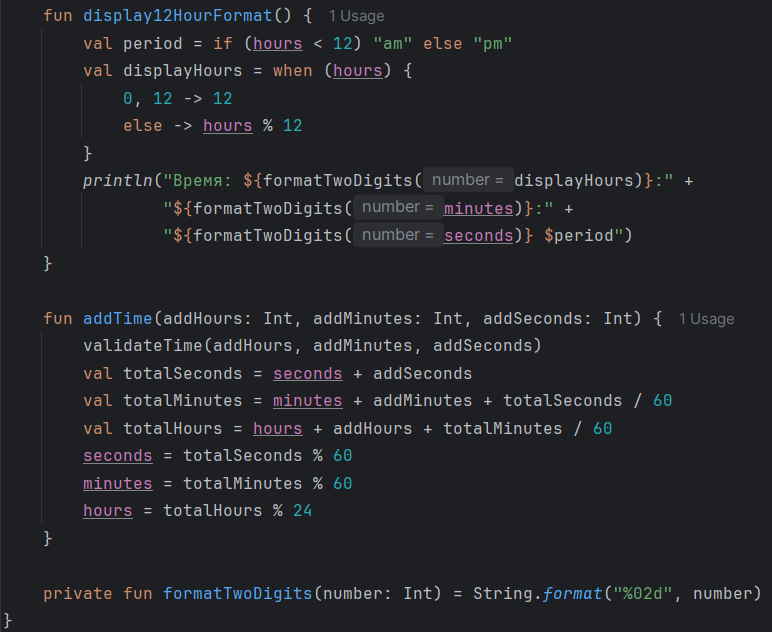


Рисунок 6.2 – Код класса TimeMeasure (часть 2)

Результат работы программы представлен на рисунке 6.3

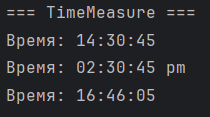


Рисунок 6.3 – Результат выполнения программы для задания 1

## 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.)

Реализован класс UniqueString, хранящий данные в виде массива символов и предоставляющий два конструктора для инициализации из массива char и из String, а также набор методов для работы со строкой: getChar() для безопасного доступа к символам с валидацией индекса, length() для получения длины, contains() с двумя перегрузками для поиска подстрок, trimLeadingSpaces() для удаления начальных пробелов через преобразование в String и обратно, и reverse() для разворота строки с использованием встроенной функции массива.

Код реализации алгоритма программы представлен на рисунке 6.4

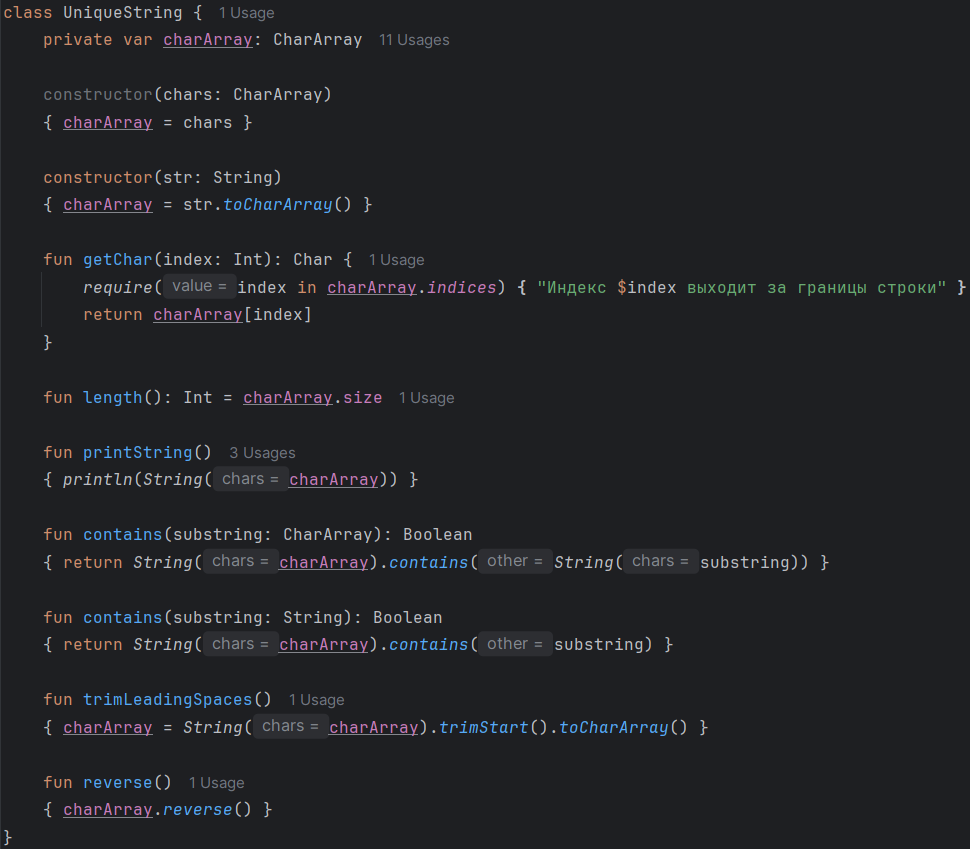


Рисунок 6.4 – Код класса UniqueString

Результат работы программы представлен на рисунке 6.5

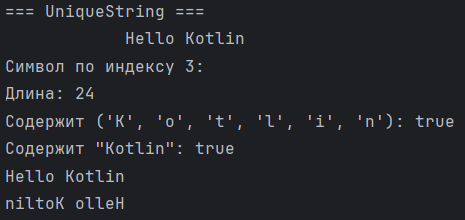


Рисунок 6.5 – Результат работы программы

## Дополнительное 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),

Cнятие 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.

Создан класс BankAccount в отдельном пакете banking с двумя конструкторами и строгой системой контроля доступа: свойство balance имеет приватное поле \_balance с кастомным геттером для округления и запретом на прямое изменение, ownerName защищен сеттером с проверкой минимальной длины, а также реализованы методы deposit() и withdraw() с валидацией сумм и автоматическим учетом операций через приватный метод logTransaction(), при этом блок init гарантирует неотрицательность начального баланса, а метод getAccountInfo() предоставляет форматированную информацию о состоянии счета.

Код реализации алгоритма программы представлен на рисунках 6.6-6.7

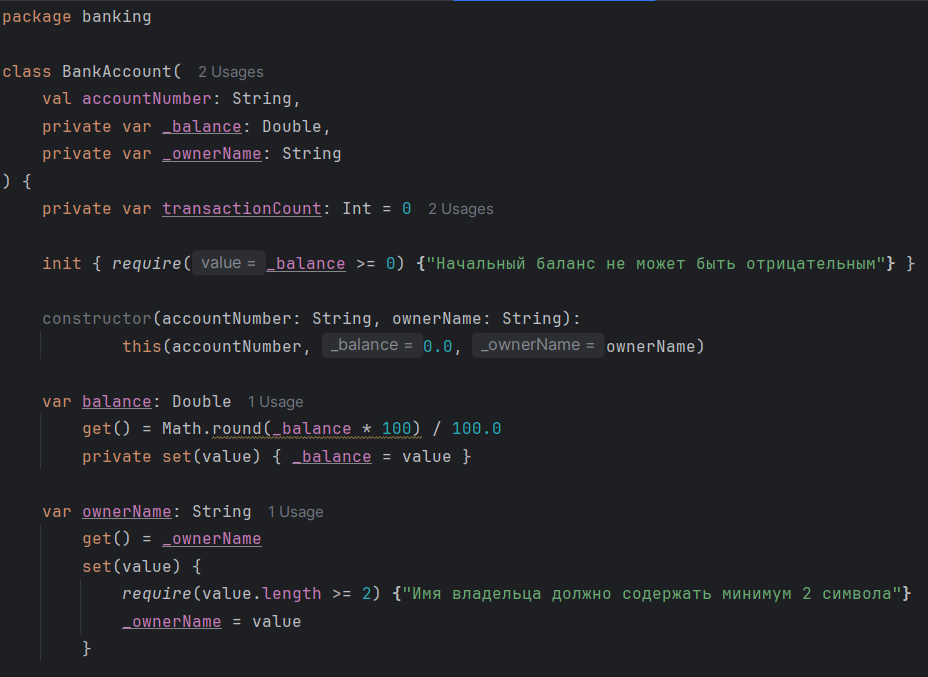


Рисунок 6.6 – Код класса BankAccount (часть 1)

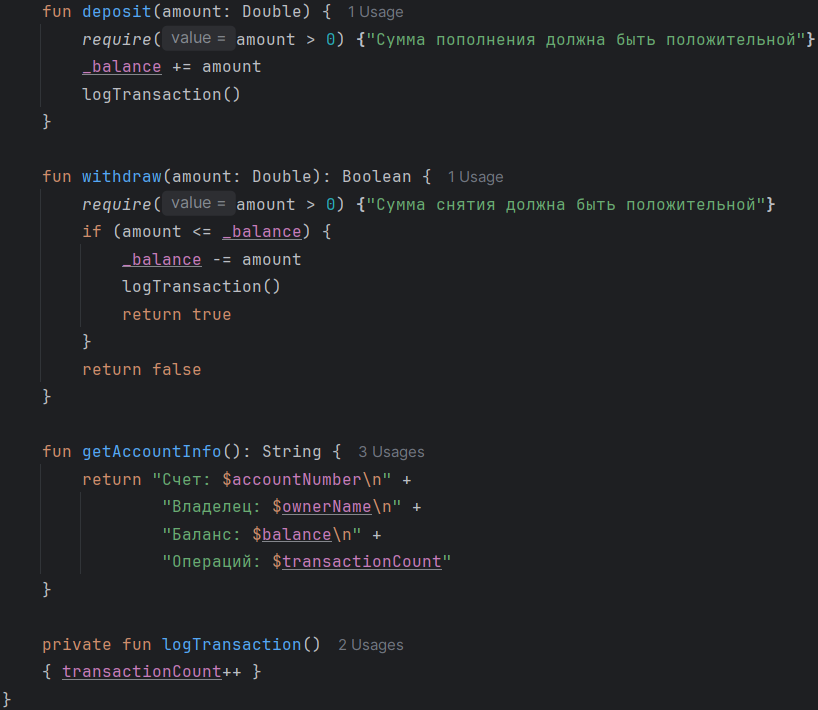


Рисунок 6.7 – Код класса BankAccount (часть 2)

Результат работы программы представлен на рисунке 6.8

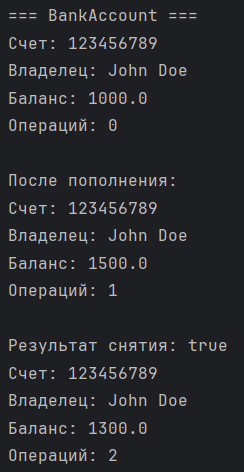


Рисунок 6.8 – Результат работы программы

## Вывод

В ходе выполнения шестой практической работы были успешно освоены продвинутые аспекты объектно-ориентированного программирования в Kotlin, включая создание классов с множественными конструкторами и организацию сложной логики инициализации с валидацией, реализацию пользовательских свойств с кастомными геттерами и сеттерами для обеспечения контроля данных, работу с пакетами для организации структуры проекта, а также применение принципов инкапсуляции для создания надежных и безопасных классов, что позволило разработать комплексную систему для работы со временем, строками и банковскими операциями с полным циклом валидации и обработки данных.

# Практическая работа 7

## Цель работы

Освоить принципы объектно-ориентированного программирования в Kotlin, включая создание абстрактных классов, наследование, использование enum классов, data-классов, а также реализацию кастомных геттеров и финальных методов.

## Решение

**Добавить тестовые данные для проверки разработанного функционала.**

**Вариант 2. Система заказов в кафе**

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.

Реализован абстрактный класс MenuItem, содержащий абстрактные свойства name и basePrice, а также абстрактный метод calculateFinalPrice(), при этом свойство id генерируется автоматически с помощью UUID и помечено как final для запрета переопределения в классах-наследниках, что обеспечивает уникальность идентификатора для каждого элемента меню.

Код реализации алгоритма программы представлен на рисунке 7.1

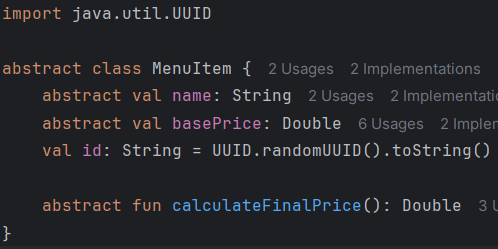


Рисунок 7.1 – Код абстрактного класса MenuItem

Результат работы программы представлен на рисунке 7.2



Рисунок 7.2 – Результат выполнения программы для части 1

## Часть 2

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

Создан data-класс Ingredient с двумя свойствами: name для названия ингредиента и isAllergen для указания аллергенности, что позволяет автоматически получать реализации методов equals(), hashCode(), toString() и copy() благодаря использованию модификатора data.

Код реализации алгоритма программы представлен на рисунке 7.3

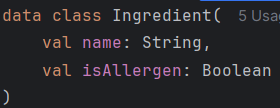


Рисунок 7.3 – Код data-класса Ingredient

Результат работы программы представлен на рисунке 7.4



Рисунок 7.4 – Результат выполнения программы для части 2

## 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).

Реализован класс Drink, наследующий от MenuItem, с переопределением свойств name и basePrice, а также добавлением свойства size типа enum class Size, при этом метод calculateFinalPrice() переопределен с логикой расчета цены в зависимости от размера порции и помечен как final для запрета дальнейшего переопределения, а также добавлен пользовательский метод toString() для красивого форматированного вывода информации о напитке.

Код реализации алгоритма программы представлен на рисунке 7.5

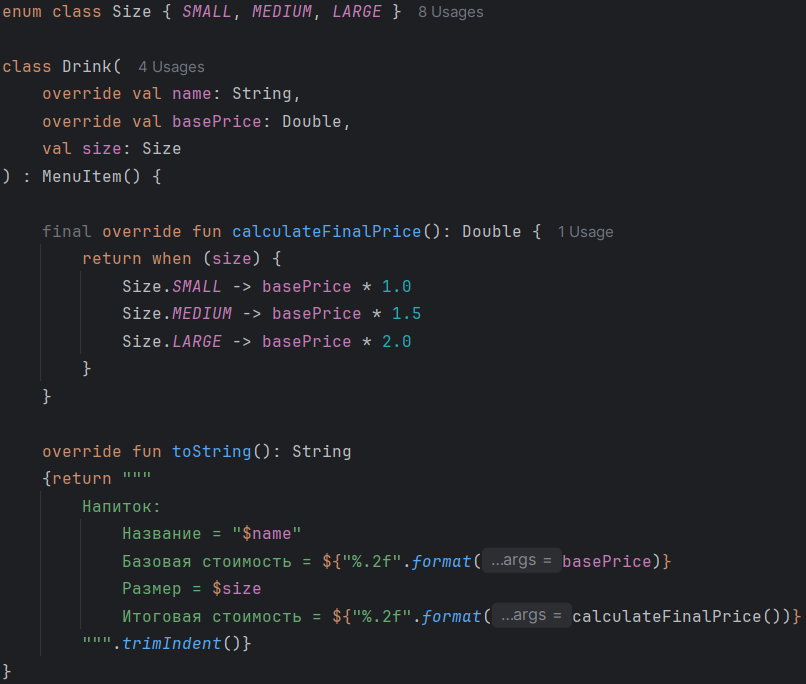


Рисунок 7.5 – Код класса Drink и enum Size

Результат работы программы представлен на рисунке 7.6

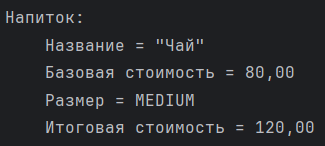


Рисунок 7.6 – Результат выполнения программы для части 3

## Часть 4

1. Создайте класс *Food*, наследующийся от *MenuItem*.
2. Переопределите свойства *name* и *basePrice*.
3. Добавьте свойство *ingredients*: *List<Ingredient>*, 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*.

Создан класс Food, наследующий от MenuItem, с переопределением свойств name и basePrice, добавлением свойства ingredients для хранения списка ингредиентов и кастомного геттера isVegetarian, который проверяет отсутствие аллергенных ингредиентов с помощью функции none(), а метод calculateFinalPrice() возвращает базовую цену без изменений, также переопределен метод toString() для вывода полной информации о блюде включая список ингредиентов и статус вегетарианства.

Код реализации алгоритма программы представлен на рисунке 7.7

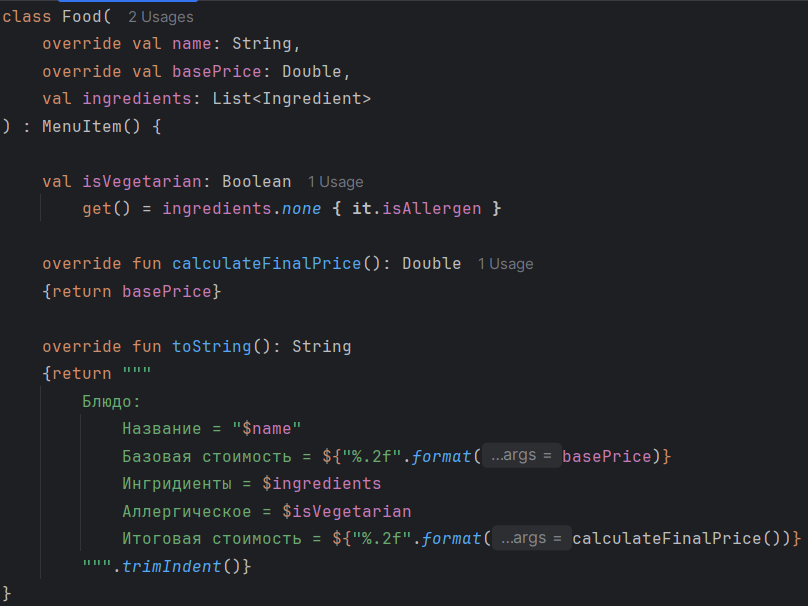


Рисунок 7.7 – Код класса Food

Результат работы программы представлен на рисунке 7.8

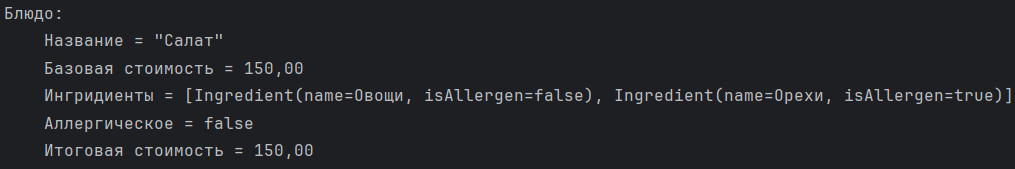


Рисунок 7.8 – Результат выполнения программы для части 4

## Вывод

В ходе выполнения седьмой практической работы были успешно освоены ключевые аспекты объектно-ориентированного программирования в Kotlin, включая создание абстрактных классов и наследование, использование enum классов для типизированных констант, применение data-классов для моделирования данных с автоматической генерацией методов, реализацию кастомных геттеров с бизнес-логикой и финальных методов для ограничения дальнейшего переопределения, что позволило построить стройную иерархию классов для системы заказов в кафе с правильным разделением ответственности и гибкостью для дальнейшего расширения функциональности.

# Практическая работа 8

## Цель работы

Освоить работу с enum классами в Kotlin, включая добавление свойств и методов, а также научиться находить и исправлять типичные ошибки в объектно-ориентированном коде, связанные с наследованием, модификаторами доступа и синтаксисом data-классов.

## Решение

## 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.

Реализован enum класс DrinkType с параметрами для хранения отображаемого названия, объема и температуры напитка, а также методы getFormattedName() для капитализации первой буквы названия, getVolume() для возврата объема и isHot() для проверки температуры выше 60 градусов, при этом в основной функции продемонстрирован перебор всех элементов enum с выводом полной информации о каждом напитке.

Код реализации алгоритма программы представлен на рисунке 8.1



Рисунок 8.1 – Код enum класса DrinkType

Результат работы программы представлен на рисунке 8.2

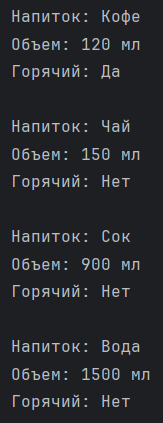


Рисунок 8.2 – Результат выполнения программы для задания 1

## Задание 2

2.1 Исправьте ошибки в данном коде: <https://clck.ru/3PpYPD>.

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 задаче исправлена сложная система банковских карт с устранением 28 различных ошибок, включая добавление модификатора open для методов наследования, корректировку логики расчета баланса, исправление модификаторов доступа к свойствам, устранение проблем с инициализацией коллекций и добавление правильной валидации данных. Во второй задаче добавлен модификатор open к методу speak() в базовом классе Animal для разрешения переопределения. В третьей задаче добавлены модификаторы val к параметрам data-класса User для корректной работы автоматически генерируемых методов.

Код реализации алгоритма программы представлен на рисунках 8.3-8.11

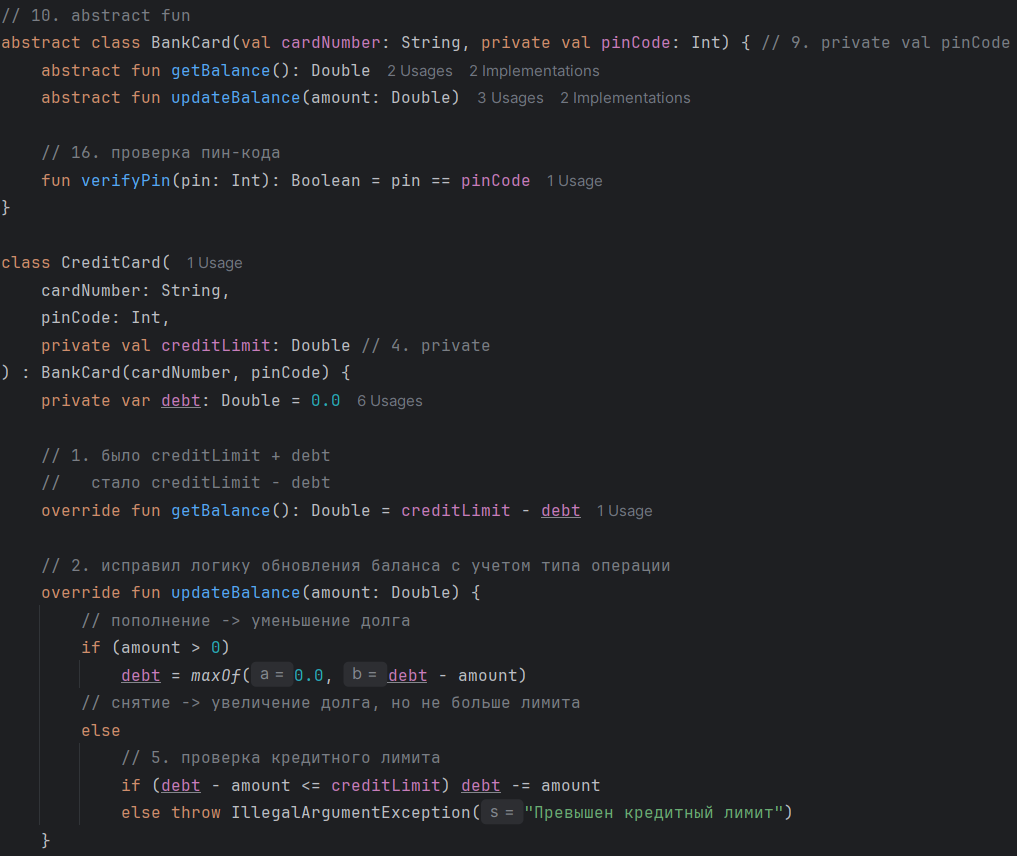


Рисунок 8.3 – Исправленный код 2.1 (часть 1)

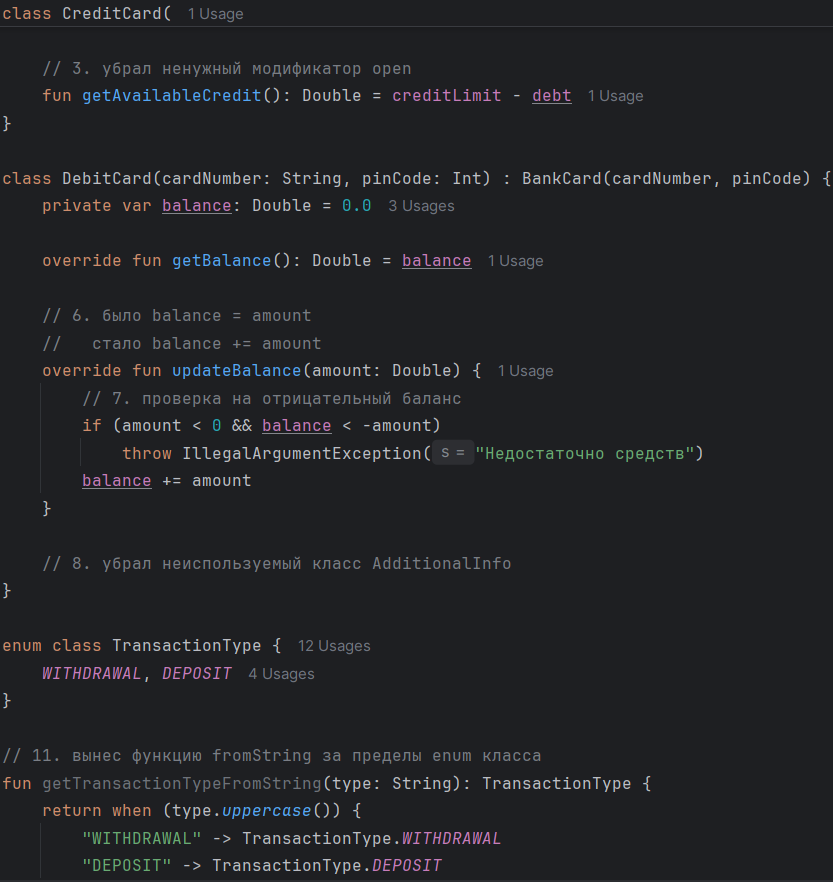


Рисунок 8.4 – Исправленный код 2.1 (часть 2)

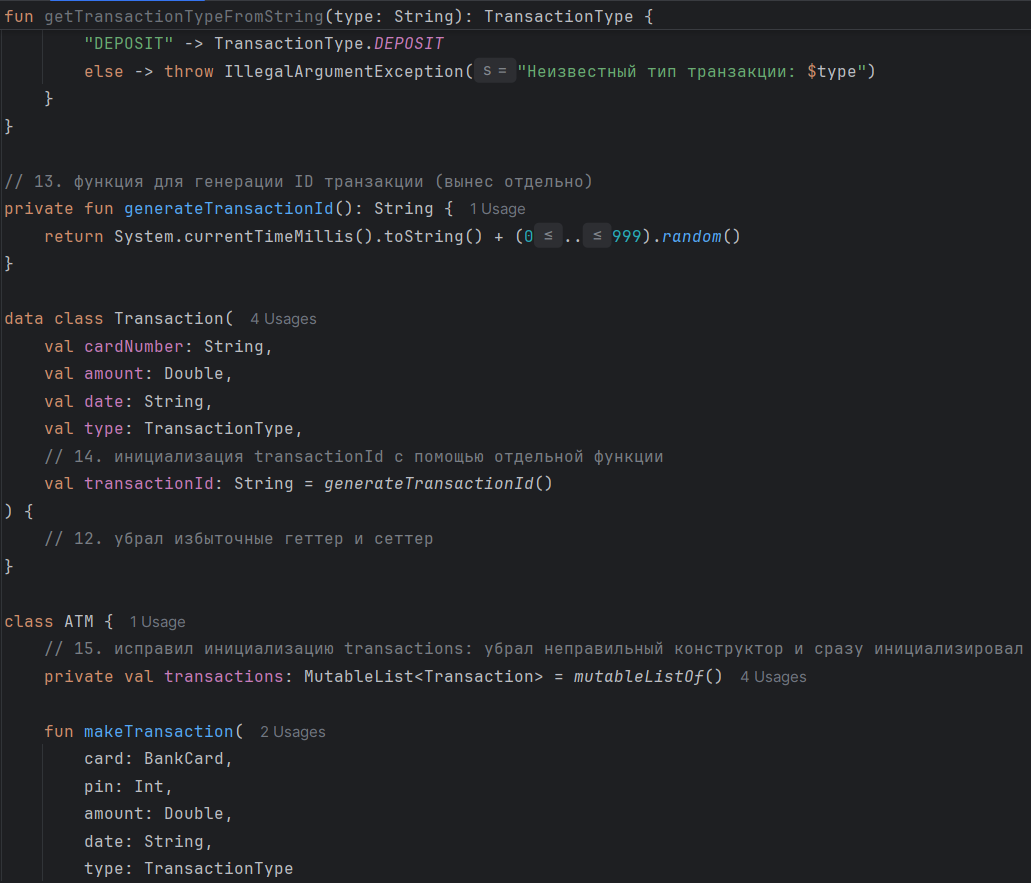


Рисунок 8.5 – Исправленный код 2.1 (часть 3)

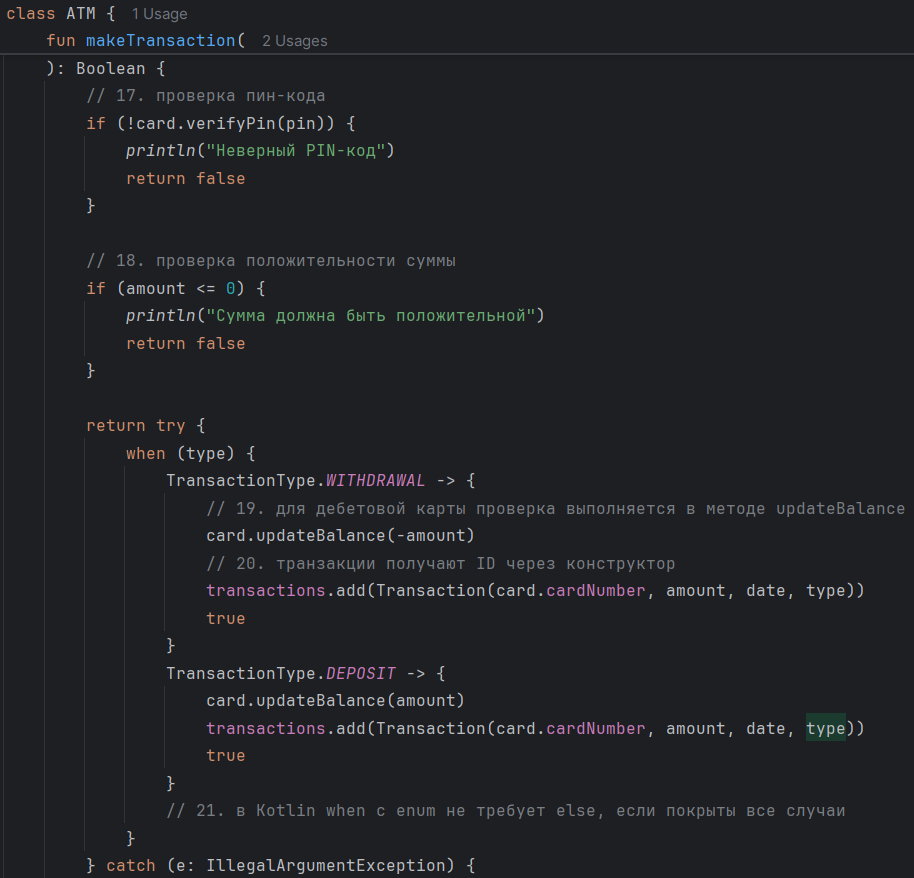


Рисунок 8.6 – Исправленный код 2.1 (часть 4)

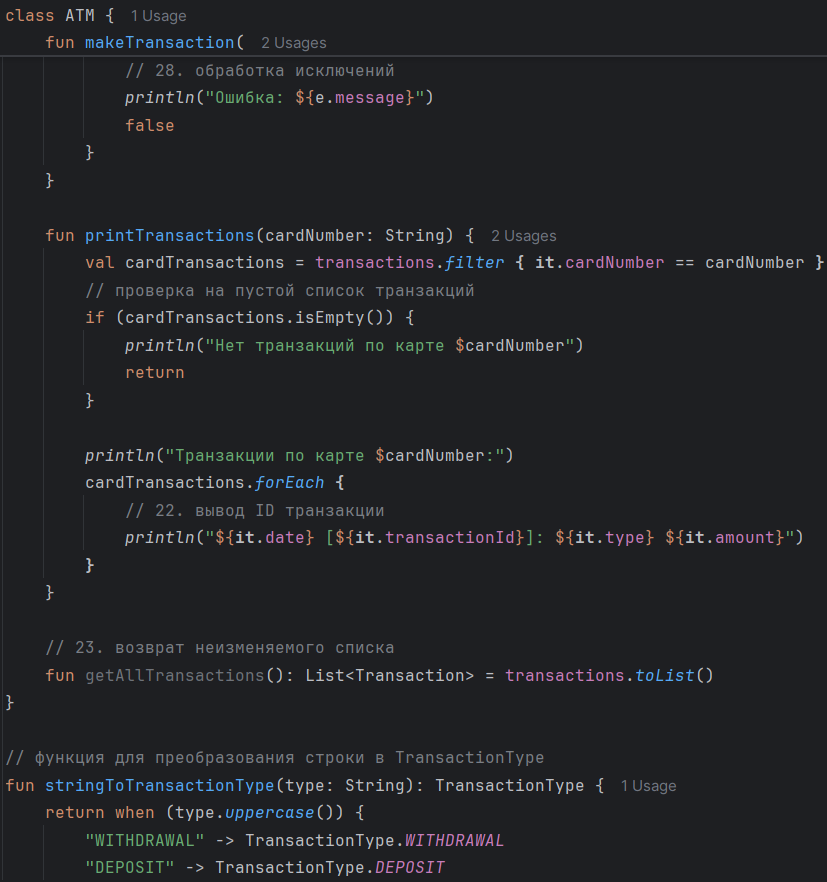


Рисунок 8.7 – Исправленный код 2.1 (часть 5)

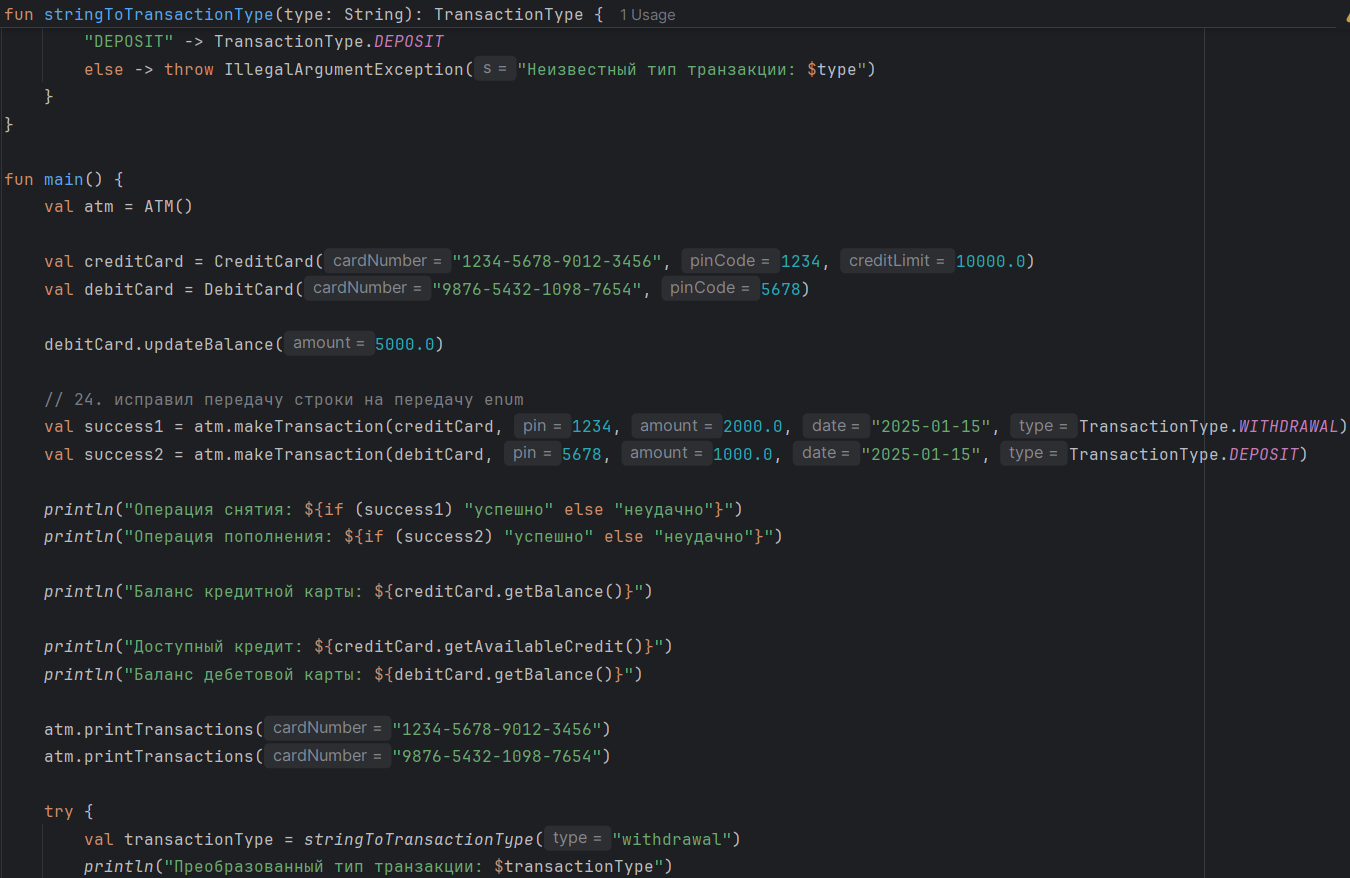


Рисунок 8.8 – Исправленный код 2.1 (часть 6)

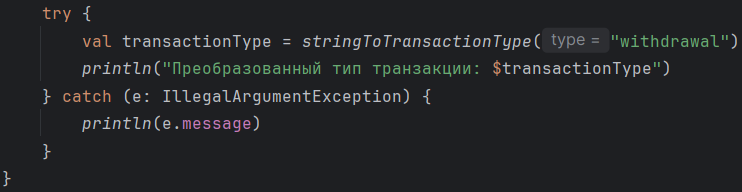


Рисунок 8.9 – Исправленный код 2.1 (часть 7)



Рисунок 8.10 – Исправленный код 2.2

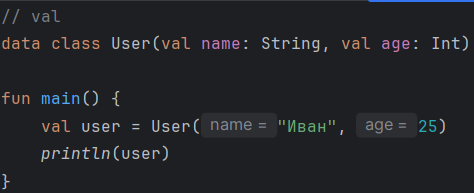


Рисунок 8.11 – Исправленный код 2.3

Результат работы программы представлен на рисунках 8.12-8.14

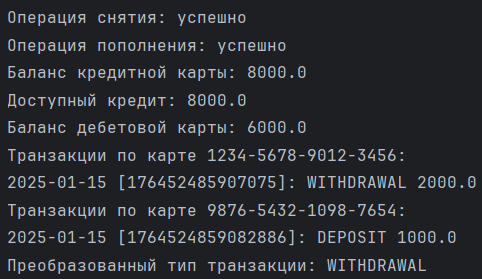


Рисунок 8.12 – Результат выполнения программы 2.1 для задания 2



Рисунок 8.13 – Результат выполнения программы 2.2 для задания 2



Рисунок 8.14 – Результат выполнения программы 2.3 для задания 2

## Вывод

В ходе выполнения восьмой практической работы были успешно освоены принципы работы с enum классами в Kotlin, включая добавление свойств и методов к элементам перечисления, а также развиты навыки отладки и исправления ошибок в объектно-ориентированном коде, что позволило идентифицировать и устранить разнообразные проблемы, связанные с наследованием, модификаторами доступа, логикой бизнес-процессов и синтаксисом data-классов, продемонстрировав глубокое понимание системы типов и принципов ООП в Kotlin.

# Практическая работа 9

## Цель работы

Освоить принципы построения сложных иерархий классов с использованием интерфейсов и абстрактных классов в Kotlin. Изучить организацию системы состояний объектов и реализацию интерактивного меню для управления поведением объектов.

## Решение

## 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.

Реализована сложная система наследования с использованием интерфейсов для разделения поведения: Birthable для способов размножения (MammalBirth, FishBirth, BirdBirth) и Movable для способов передвижения (Flying, Swimming). Абстрактный класс Animal содержит общее состояние и базовые методы, а конкретные классы животных комбинируют интерфейсы для создания уникального поведения, при этом для изменения стандартного поведения интерфейсов используются анонимные объекты. Система меню предоставляет полный контроль над состояниями животных с возможностью кормления, сна, игр и просмотра характеристик.

Код реализации алгоритма программы представлен на рисунках 9.1-9.6

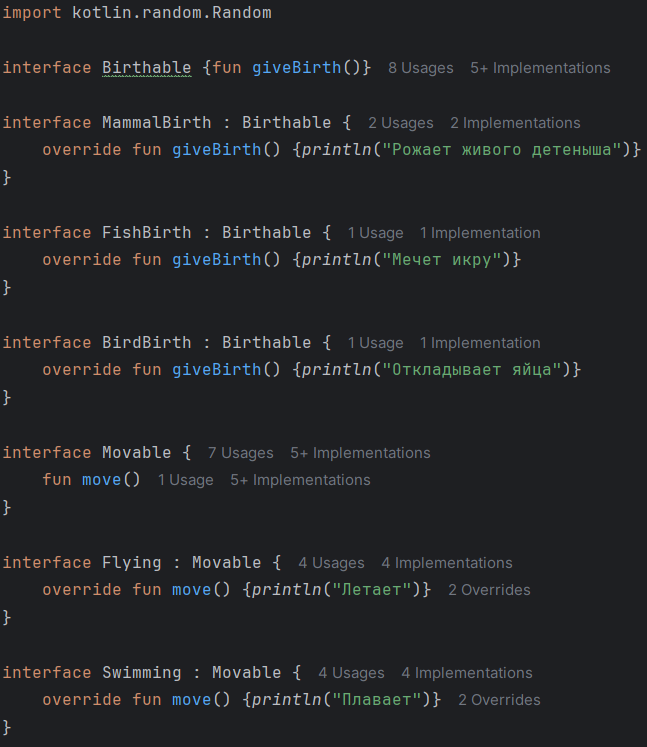


Рисунок 9.1 – Код системы управления животными (часть 1)

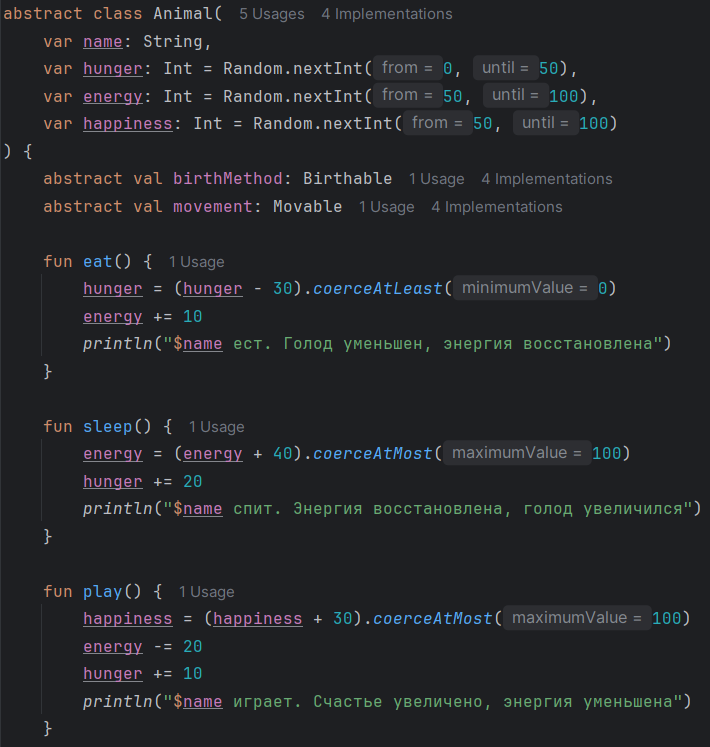


Рисунок 9.2 – Код системы управления животными (часть 2)

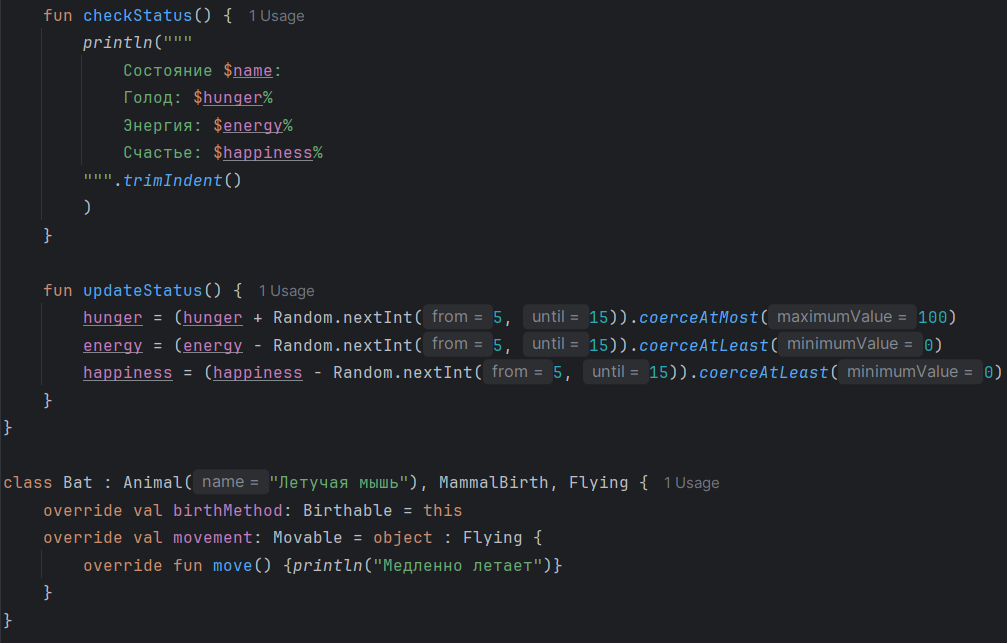


Рисунок 9.3 – Код системы управления животными (часть 3)

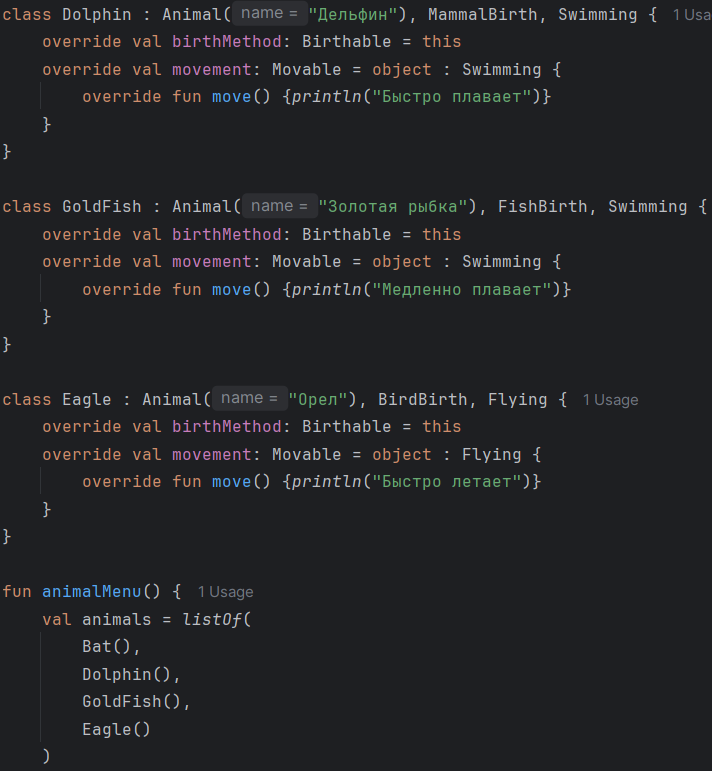


Рисунок 9.4 – Код системы управления животными (часть 4)



Рисунок 9.5 – Код системы управления животными (часть 5)

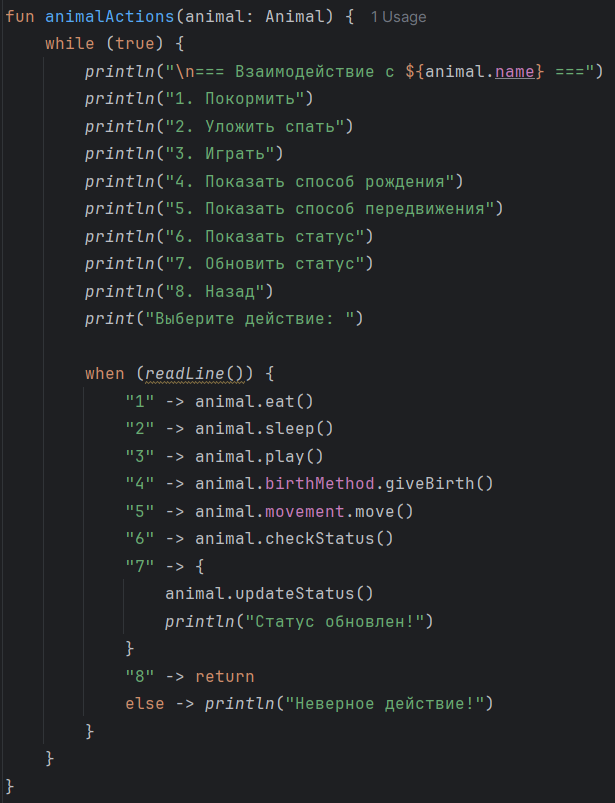


Рисунок 9.6 – Код системы управления животными (часть 6)

Результат работы программы представлен на рисунках 9.7-9.9

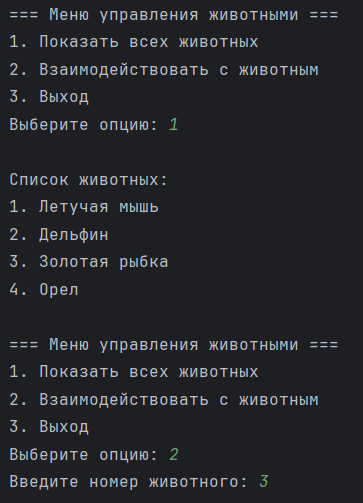


Рисунок 9.7 – Результат выполнения программы для задания 1 (часть 1)

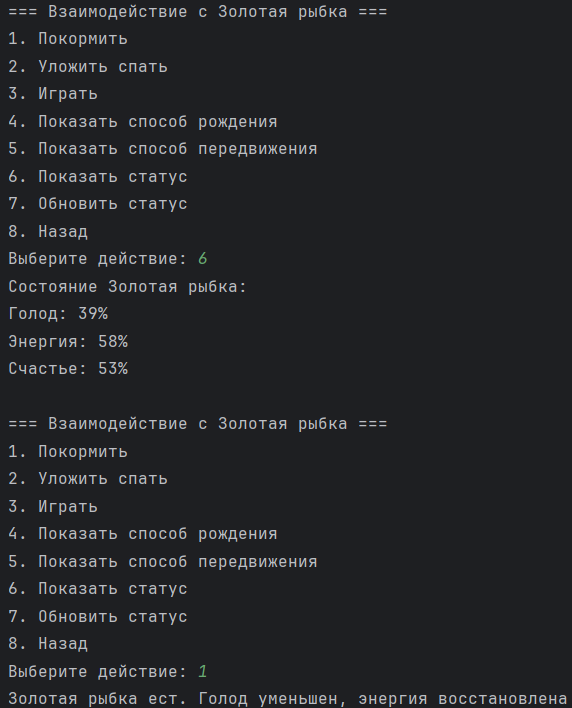


Рисунок 9.8 – Результат выполнения программы для задания 1 (часть 2)

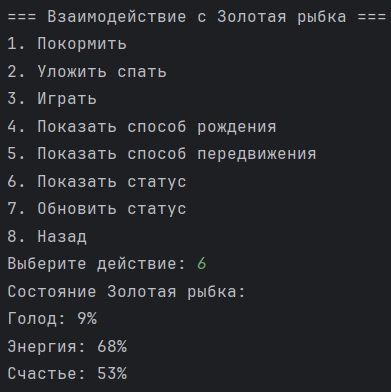


Рисунок 9.9 – Результат выполнения программы для задания 1 (часть 3)

## 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.

Создана простая иерархия устройств с интерфейсом Device в качестве маркера и классами Computer, Phone и OtherDevice. Класс BestRepairEver содержит метод canRepair(), который использует выражение when с интеллектуальным приведением типов для проверки поддерживаемых типов устройств, возвращая true для Computer и Phone, и false для всех остальных устройств, что демонстрирует принцип работы с типами через проверку во время выполнения.

Код реализации алгоритма программы представлен на рисунке 9.10



Рисунок 9.10 – Код системы мастерской ремонта

Результат работы программы представлен на рисунке 9.11

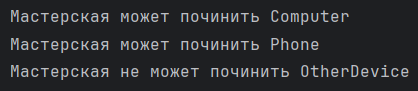


Рисунок 9.11 – Результат выполнения программы для задания 2

## Вывод

В ходе выполнения девятой практической работы были успешно освоены принципы построения сложных объектно-ориентированных систем с использованием множественного наследования через интерфейсы, реализована гибкая архитектура с разделением ответственности между различными типами поведения, создана система управления состояниями объектов с динамическим обновлением характеристик, а также разработано интерактивное консольное меню для взаимодействия с объектами, что продемонстрировало умение проектировать и реализовывать сложные программные системы с четкой структурой и удобным пользовательским интерфейсом.

# Практическая работа 10

## Цель работы

Освоить механизмы обработки исключений и создания пользовательских классов исключений в Kotlin, а также изучить работу с анонимными объектами и коллекциями для решения практических задач валидации данных и управления пользовательскими списками.

## Решение

## 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.

Создан класс FormValidator с набором методов для проверки различных полей анкеты: validateName() проверяет длину имени и заглавную первую букву, validateBirthdate() использует DateTimeFormatter для парсинга даты и проверяет ее нахождение в допустимом диапазоне, validateGender() проверяет соответствие перечислению Gender через valueOf(), а validateWeight() обрабатывает преобразование строки в число и проверяет положительность значения. Для каждого типа ошибок созданы соответствующие классы исключений, наследуемые от Exception, что позволяет точно идентифицировать тип ошибки при валидации.

Код реализации алгоритма программы представлен на рисунках 10.1-10.3

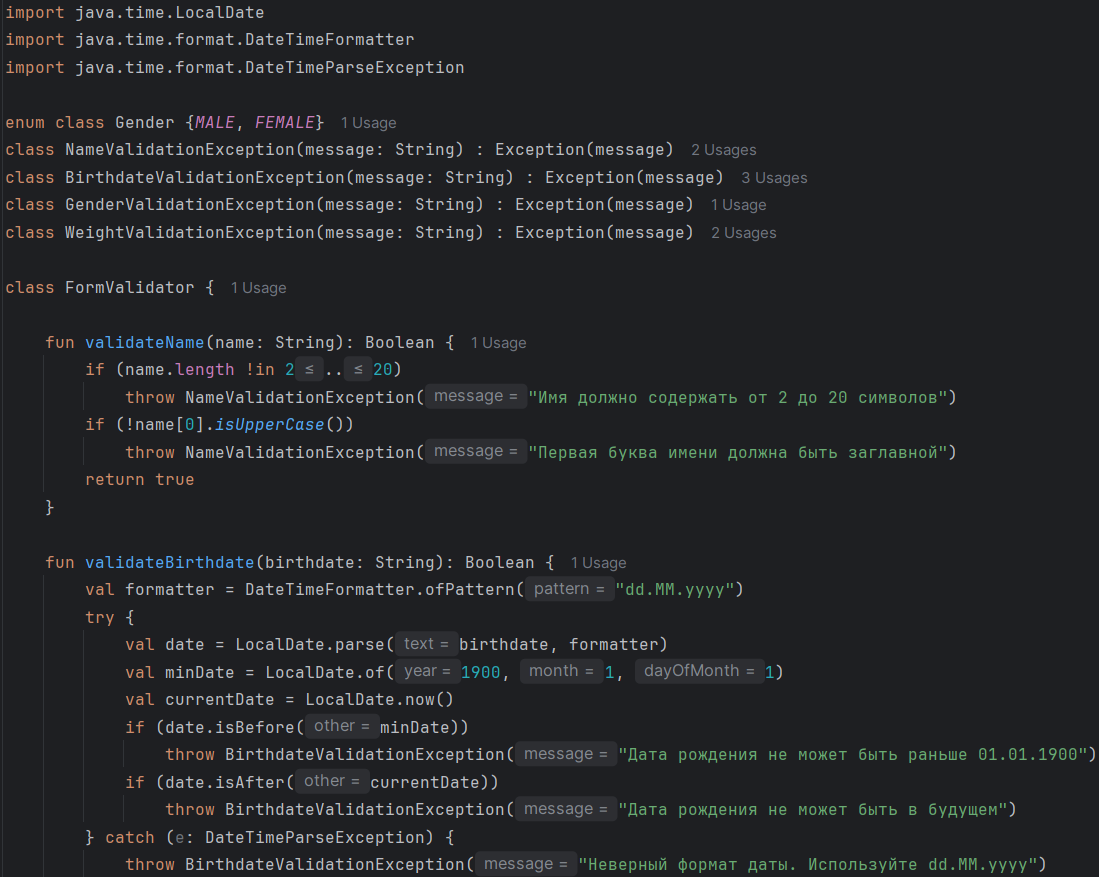


Рисунок 10.1 – Код класса FormValidator (часть 1)

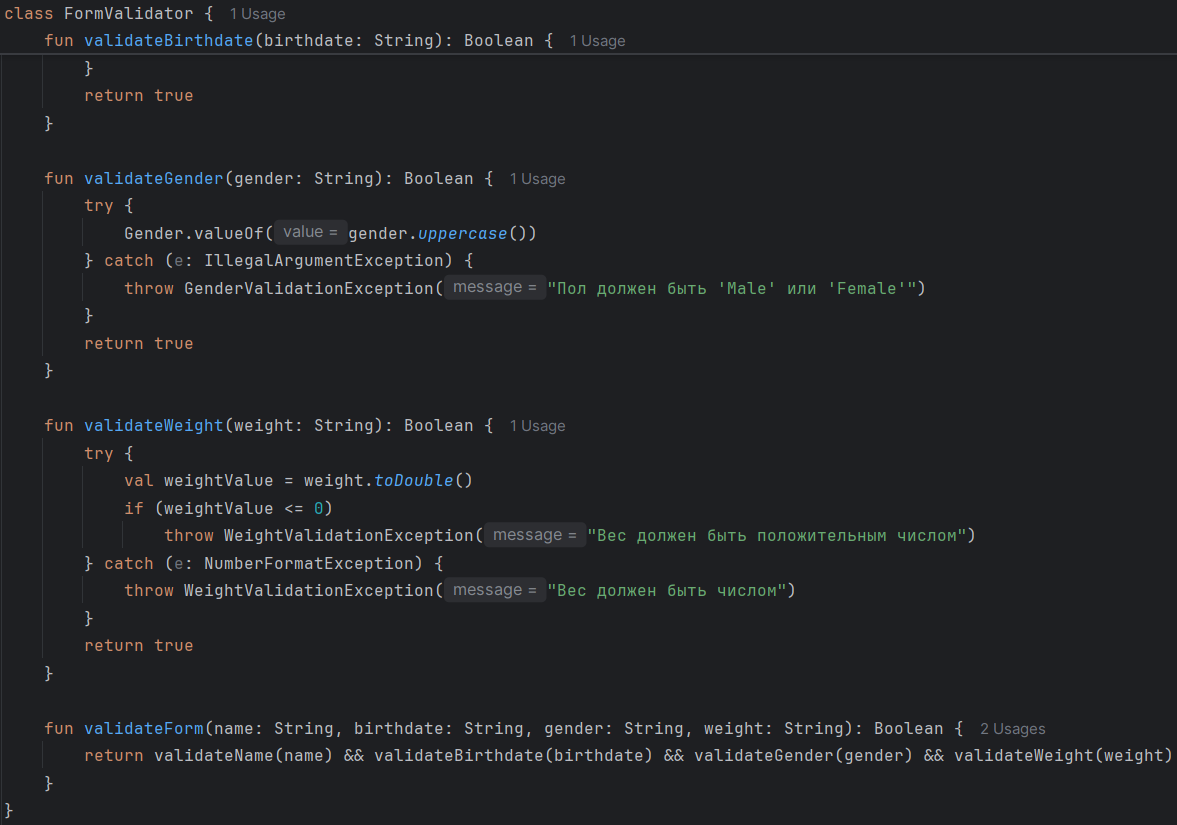


Рисунок 10.2 – Код класса FormValidator (часть 2)

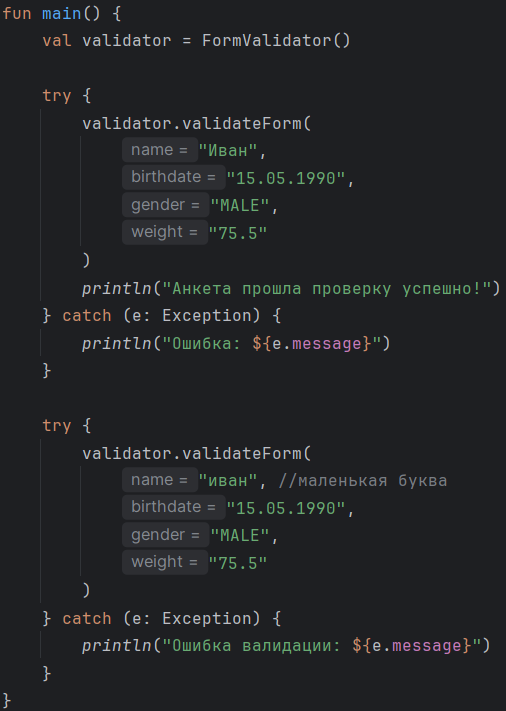


Рисунок 10.3 – Код класса FormValidator (часть 3)

Результат работы программы представлен на рисунке 10.4

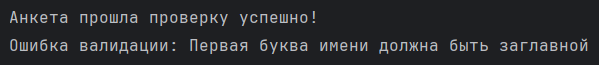


Рисунок 10.4 – Результат выполнения программы для задания 1

## 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.

Реализован анонимный объект UserList с использованием ключевого слова object, который содержит приватную коллекцию пользователей и предоставляет методы для добавления пользователей, поиска самого старшего через maxByOrNull(), вывода информации о пользователях и расчета статистических показателей. Data-класс User инкапсулирует информацию о пользователе, включая список друзей, что позволяет демонстрировать работу с вложенными структурами данных и агрегирующими функциями коллекций Kotlin.

Код реализации алгоритма программы представлен на рисунках 10.5-10.6

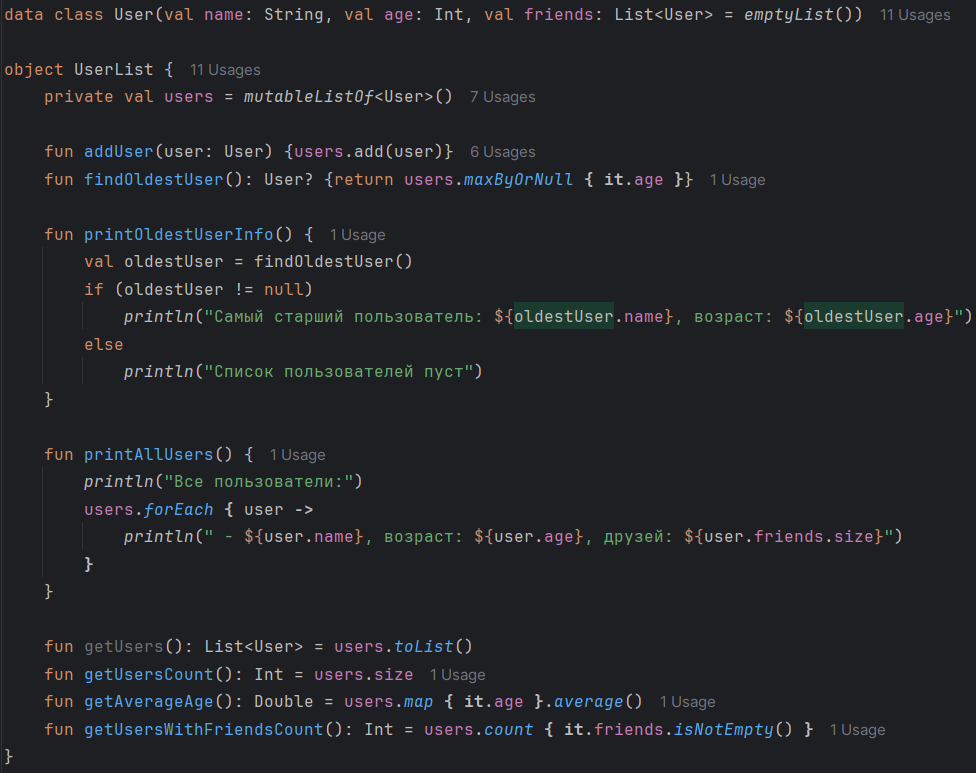


Рисунок 10.5 – Код анонимного объекта UserList (часть 1)



Рисунок 10.6 – Код анонимного объекта UserList (часть 2)

Результат работы программы представлен на рисунке 10.7

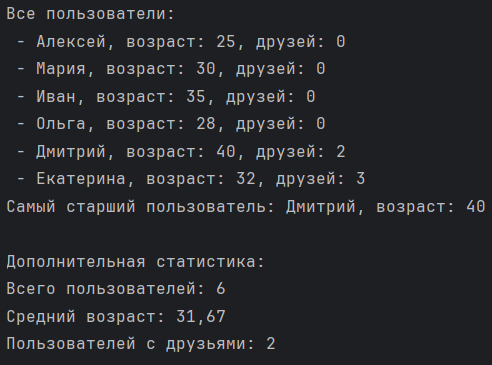


Рисунок 10.7 – Результат выполнения программы для задания 2

## Вывод

В ходе выполнения десятой практической работы были успешно освоены механизмы обработки исключений в Kotlin, включая создание пользовательских классов исключений для точной идентификации ошибок валидации данных, а также работа с анонимными объектами для организации глобального доступа к функциональности без создания экземпляров класса. Приобретены навыки работы с датами и парсингом строк, использования перечислений для ограничения допустимых значений и применения агрегирующих функций коллекций для анализа данных, что позволило создать надежную систему валидации входных данных и эффективную систему управления пользовательскими списками со статистическим анализом.

# Практическая работа 11

## Цель работы

Освоить использование scope-функций Kotlin для удобной работы с объектами, изучить принципы обобщенного программирования с ограничениями типов, а также научиться создавать функции расширения и системы логирования с поддержкой различных типов данных.

## Решение

## Задание 1. Управление заказами в интернет-магазине

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).

Реализована система управления заказами с использованием различных scope-функций: apply() для инициализации заказа и добавления продуктов, let() для безопасной обработки заказа и изменения его статуса, run() для расчета скидки на основе общей стоимости, и also() для выполнения дополнительных действий после завершения обработки, демонстрируя практическое применение контекстных функций для последовательной обработки объектов.

Код реализации алгоритма программы представлен на рисунке 11.1

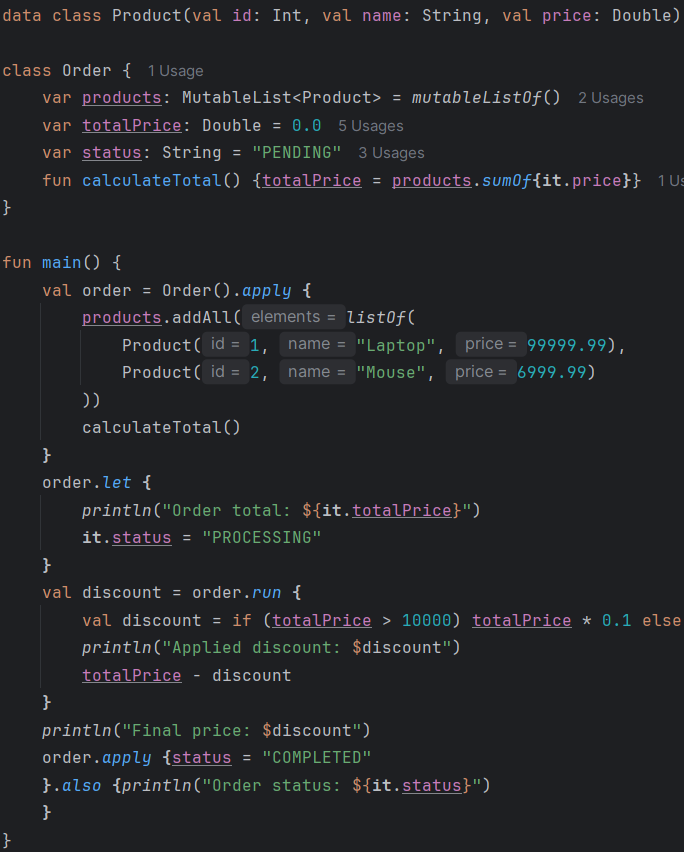


Рисунок 11.1 – Код системы управления заказами

Результат работы программы представлен на рисунке 11.2

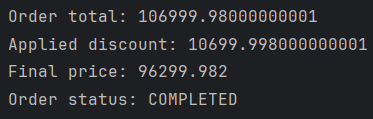


Рисунок 11.2 – Результат выполнения программы для задания 1

## Задание 2. Создание системы логирования ошибок для разных типов данных

Создайте класс ErrorLogger<T>, 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).

Создан обобщенный класс ErrorLogger с методом logError(), который использует выражение when с интеллектуальным приведением типов для определения типа ошибки и добавления соответствующего префикса, а также реализована функция расширения logAndSend(), добавляющая функциональность отправки ошибки на сервер, демонстрируя принципы работы с обобщенными типами и расширениями в Kotlin.

Код реализации алгоритма программы представлен на рисунке 11.3



Рисунок 11.3 – Код класса ErrorLogger

Результат работы программы представлен на рисунке 11.4

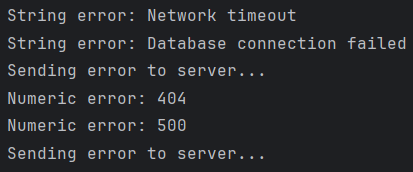


Рисунок 11.4 – Результат выполнения программы для задания 2

## Задание 3. Проверка значений с использованием ограничений обобщений

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.

Реализован обобщенный класс ValueValidator с ограничением типа T : Any, который в методе validate() проверяет значения в зависимости от их типа: для Int и Double проверяется положительность, для String проверяется длина более 5 символов, а для остальных типов выбрасывается исключение, демонстрируя работу с ограничениями обобщений и проверкой типов во время выполнения.

Код реализации алгоритма программы представлен на рисунке 11.5

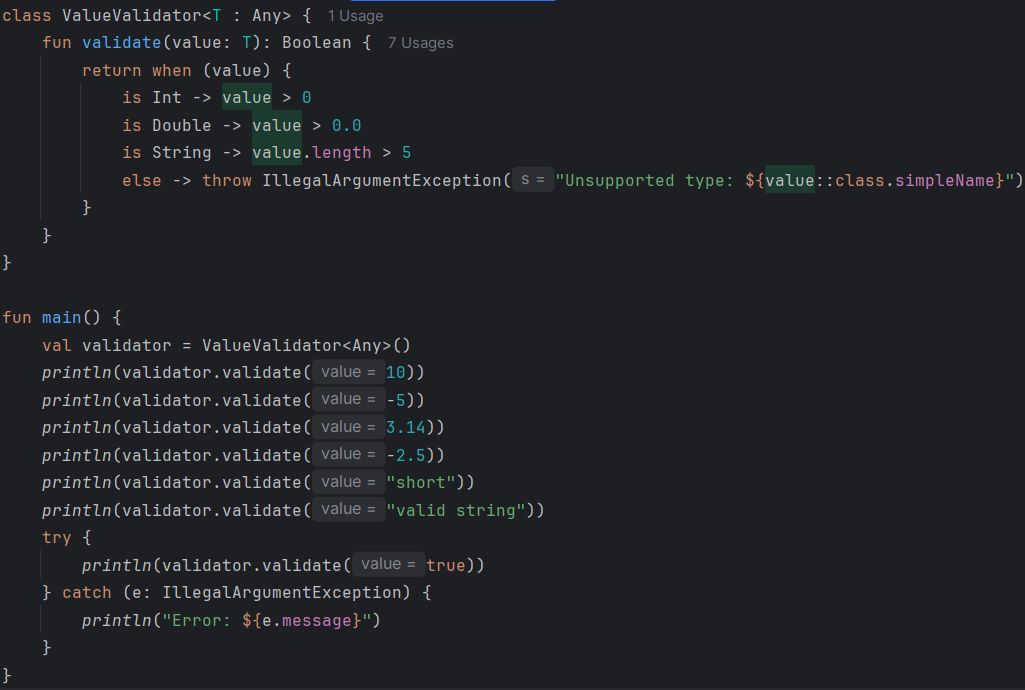


Рисунок 11.5 – Код класса ValueValidator

Результат работы программы представлен на рисунке 11.6

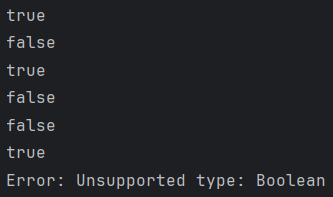


Рисунок 11.6 – Результат выполнения программы для задания 3

## **Дополнительное задание**

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

Класс ErrorLogger дополнен приватным свойством logFile, представляющим файл для записи логов, и в метод logError() добавлен вызов appendText() для записи отформатированных сообщений в файл errors.log, что обеспечивает сохранение истории ошибок между запусками программы и демонстрирует работу с файловой системой в Kotlin.

Код реализации алгоритма программы представлен на рисунке 11.7



Рисунок 11.7 – Расширенный код ErrorLogger с записью в файл

Результат работы программы представлен на рисунке 11.8

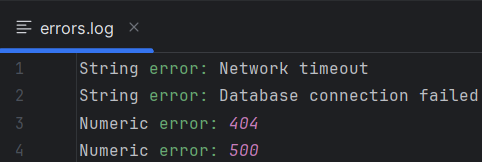


Рисунок 11.8 – Результат выполнения программы для доп. задания

## Вывод

В ходе выполнения одиннадцатой практической работы были успешно освоены ключевые возможности языка Kotlin, включая использование scope-функций для удобной работы с объектами, создание обобщенных классов с ограничениями типов для типобезопасного программирования, разработку функций расширения для добавления функциональности существующим классам, а также реализацию системы логирования с поддержкой различных типов данных и сохранением в файл, что продемонстрировало глубокое понимание современных парадигм программирования и готовность к созданию сложных, поддерживаемых приложений на языке Kotlin.

# Практическая работа 12

## Цель работы

Освоить работу с nullable типами в Kotlin, изучить безопасные способы обработки потенциально отсутствующих значений, а также научиться создавать и использовать функции расширения для добавления нового функционала к существующим типам данных.

## Решение

## Задание 1. Работа с Nullable типами

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.

Реализован класс UserProfile с nullable свойствами для различных характеристик пользователя, где метод printProfileInfo() использует оператор безопасного вызова ?. для доступа к значениям и оператор элвиса ?: для предоставления значений по умолчанию при отсутствии данных, что позволяет безопасно обрабатывать потенциально отсутствующие значения без риска возникновения NullPointerException и обеспечивает корректный вывод информации даже при частично заполненном профиле.

Код реализации алгоритма программы представлен на рисунке 12.1

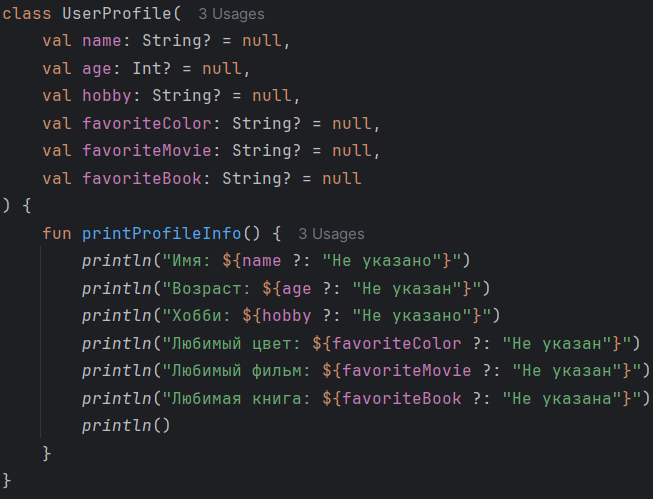


Рисунок 12.1 – Код класса UserProfile

Результат работы программы представлен на рисунке 12.2



Рисунок 12.2 – Результат выполнения программы для задания 1

## Задание 2. Преобразование типов и использование функций расширения

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:
   * Конвертируют температуру из Цельсия в Фаренгейты.
   * Конвертируют километры в мили.
   * Конвертируют килограммы в фунты.
   * и другие.
2. Реализовать обработку возможных ошибок ввода.
3. Реализуйте консольное меню для проверки функционала.

Реализован комплекс функций расширения для типа Double, охватывающих конвертацию единиц измерения в семи категориях: температура, вес, время, объем, площадь, углы и цифровые данные, где каждая функция выполняет конкретное преобразование с использованием соответствующих коэффициентов, а также создано интерактивное консольное меню с функциями валидации ввода и обработки выбора пользователя, обеспечивающее удобный интерфейс для выполнения преобразований между различными единицами измерения внутри каждой категории.

Код реализации алгоритма программы представлен на рисунках 12.3-12.8

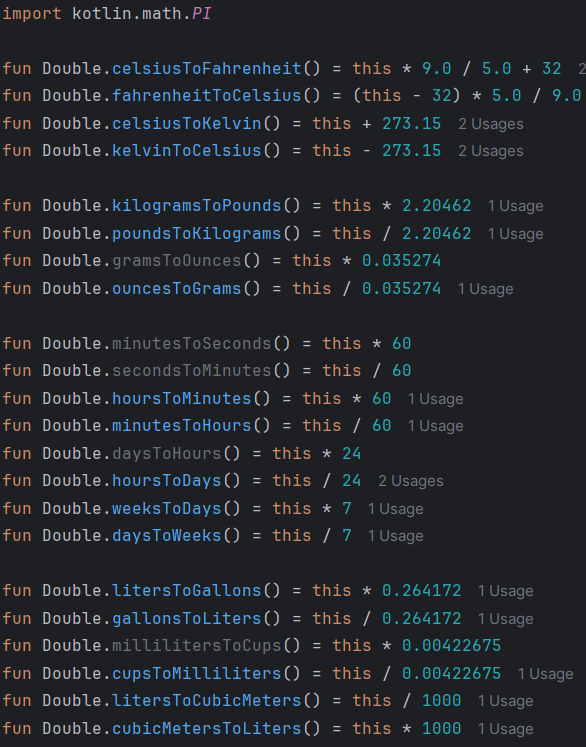


Рисунок 12.3 – Код системы конвертации единиц измерения (часть 1)

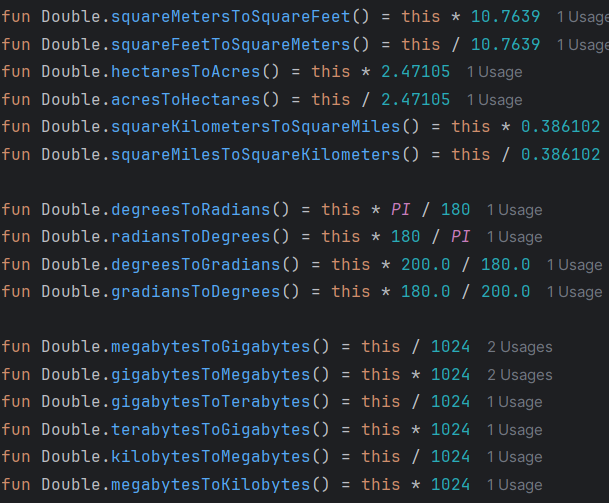


Рисунок 12.4 – Код системы конвертации единиц измерения (часть 2)



Рисунок 12.5 – Код системы конвертации единиц измерения (часть 3)

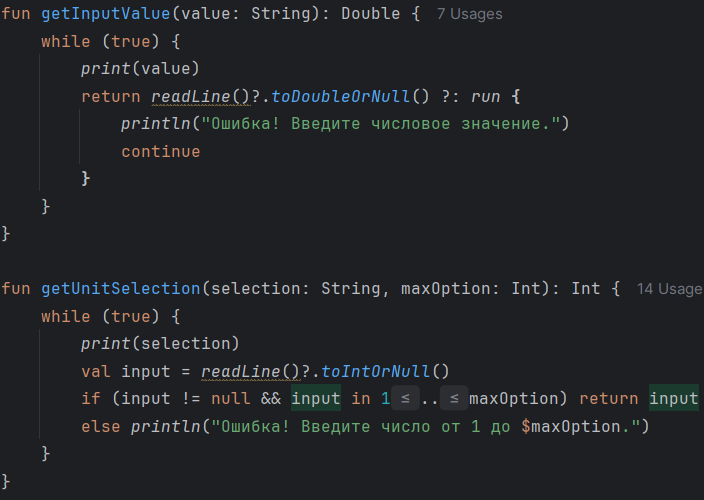


Рисунок 12.6 – Код системы конвертации единиц измерения (часть 4)

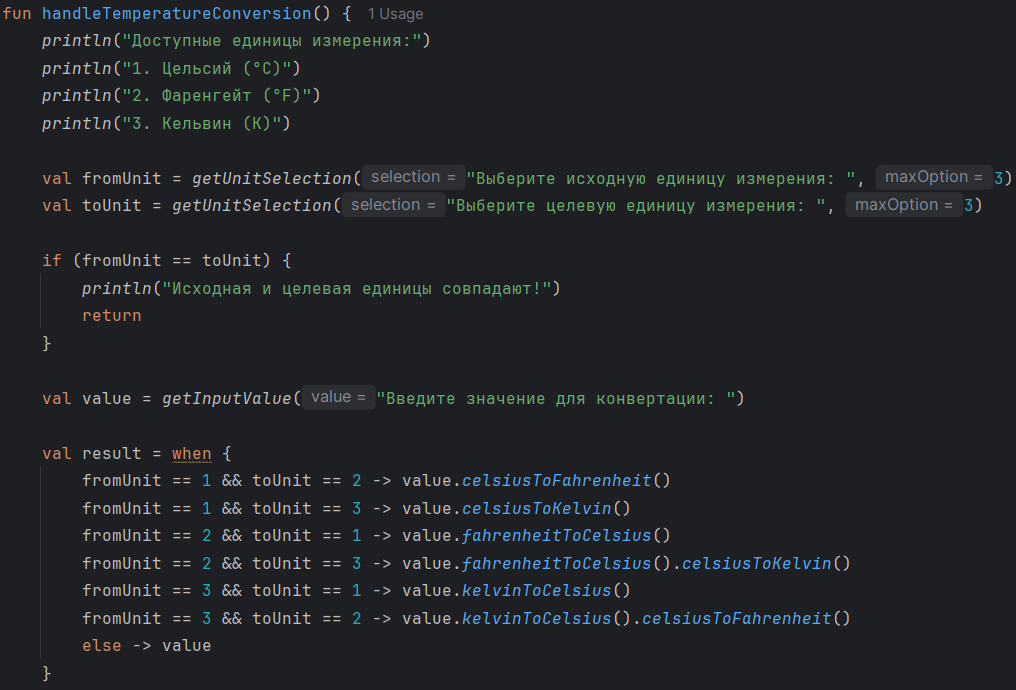


Рисунок 12.7 – Код системы конвертации единиц измерения (часть 5)



Рисунок 12.8 – Код системы конвертации единиц измерения (часть 6)

Обработчики остальных величин описаны также, но с заменой на необходимые значения.

Результат работы программы представлен на рисунке 12.9

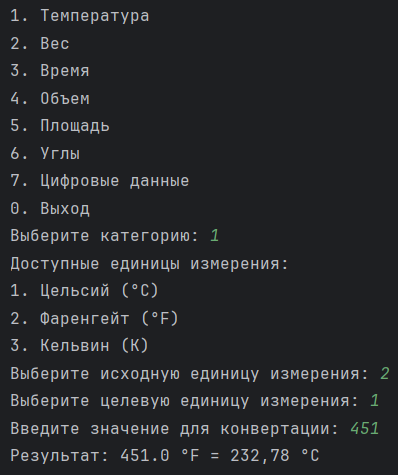


Рисунок 12.9 – Результат выполнения программы для задания 2

## Вывод

В ходе выполнения двенадцатой практической работы были успешно освоены ключевые аспекты работы с nullable типами в Kotlin, включая использование безопасных операторов для обработки потенциально отсутствующих значений и предотвращения исключений, а также изучены принципы создания и применения функций расширения для добавления нового функционала к существующим типам без модификации их исходного кода, что позволило создать надежную систему обработки пользовательских данных с поддержкой неполных профилей и разработать комплексную систему конвертации единиц измерения с модульной архитектурой и удобным пользовательским интерфейсом.

# Практическая работа 13

## Цель работы

Освоить работу с коллекциями в Kotlin, изучить методы обработки и анализа данных с использованием различных функций высшего порядка, а также научиться создавать системы для анализа бизнес-данных на примере заказов интернет-магазина и продаж.

## Решение

## Задание 1. Анализ заказов в интернет-магазине

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.

Реализован объект OrderManager с использованием изменяемой коллекции orders для хранения заказов, представленных data-классом Order, и набором методов для анализа данных: addOrder() для добавления новых заказов, removeOrdersBelowPrice() для фильтрации заказов по минимальной сумме с помощью removeAll(), getUniqueCustomers() для получения уникальных клиентов через distinct(), getTotalRevenue() для расчета общего дохода с использованием sumOf(), и getTop3MostExpensive() для получения самых дорогих заказов через сортировку и взятие первых трех элементов.

Код реализации алгоритма программы представлен на рисунке 13.1

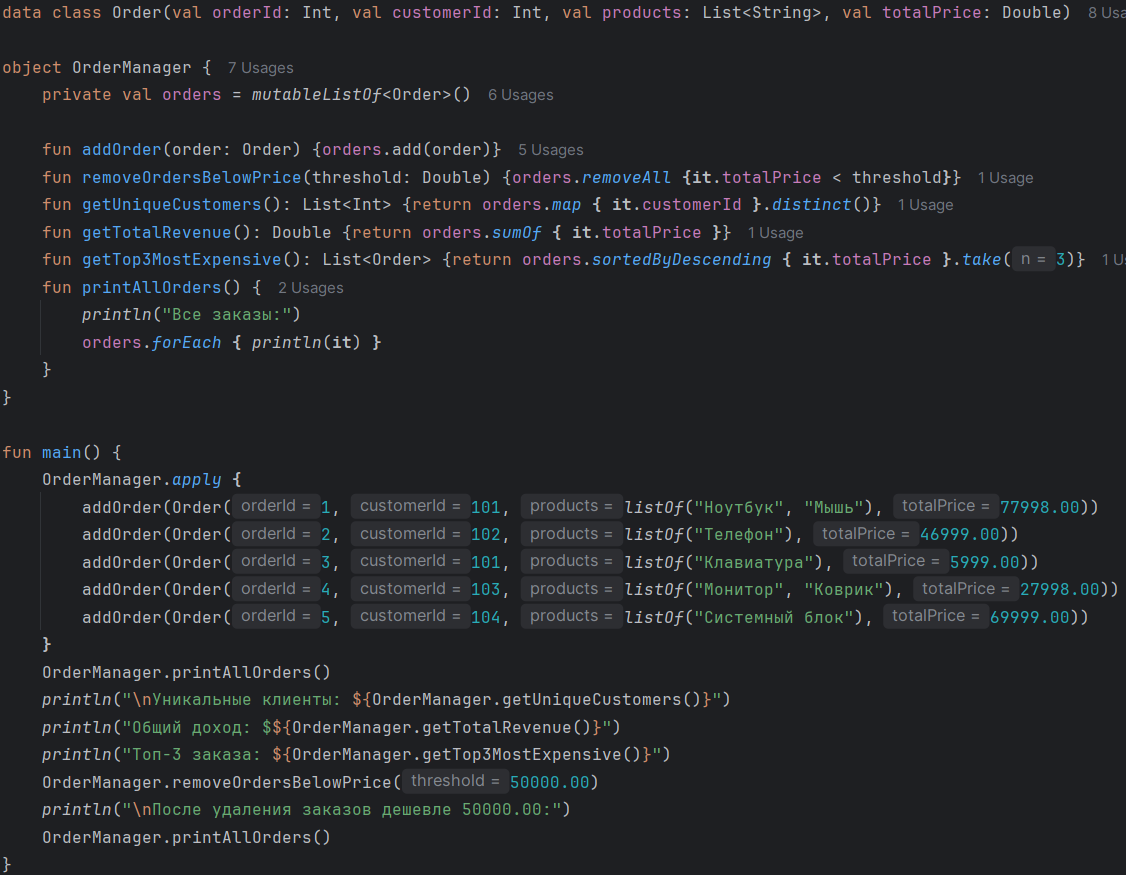


Рисунок 13.1 – Код системы анализа заказов

Результат работы программы представлен на рисунке 13.2

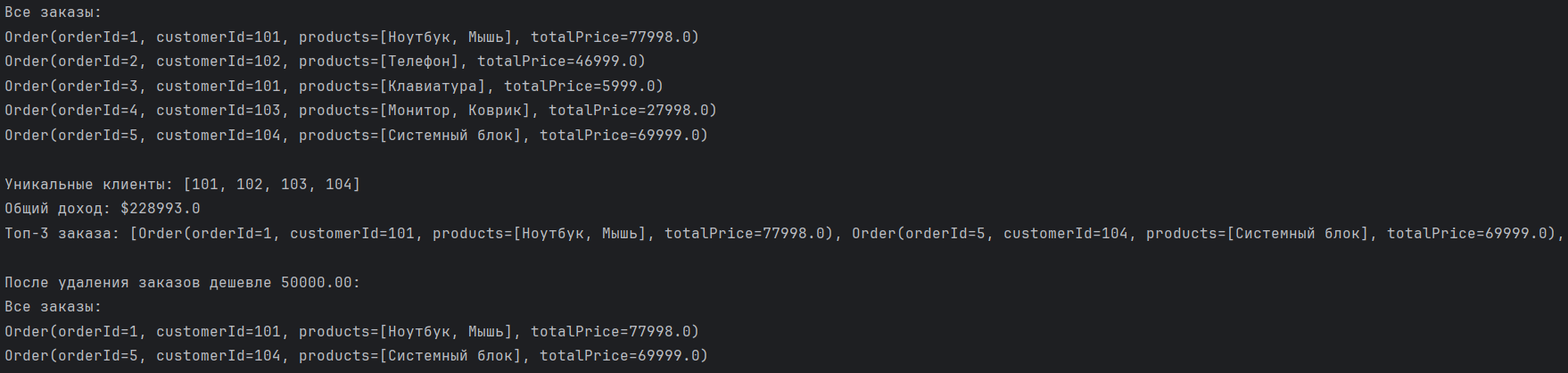


Рисунок 13.2 – Результат выполнения программы для задания 1

## Задание 2. Анализ данных о продажах

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.

Создан класс SalesAnalyzer, работающий с неизменяемой коллекцией продаж, представленных data-классом Sale, и реализующий методы анализа: getTotalRevenue() для расчета общей выручки через sumOf(), getSalesByProduct() для группировки продаж по продуктам и подсчета общего количества с помощью groupBy() и mapValues(), getSalesByMonth() для фильтрации продаж по месяцу через split() даты, и getTopProductByRevenue() для определения продукта с максимальной выручкой с использованием группировки и maxByOrNull().

Код реализации алгоритма программы представлен на рисунке 13.3

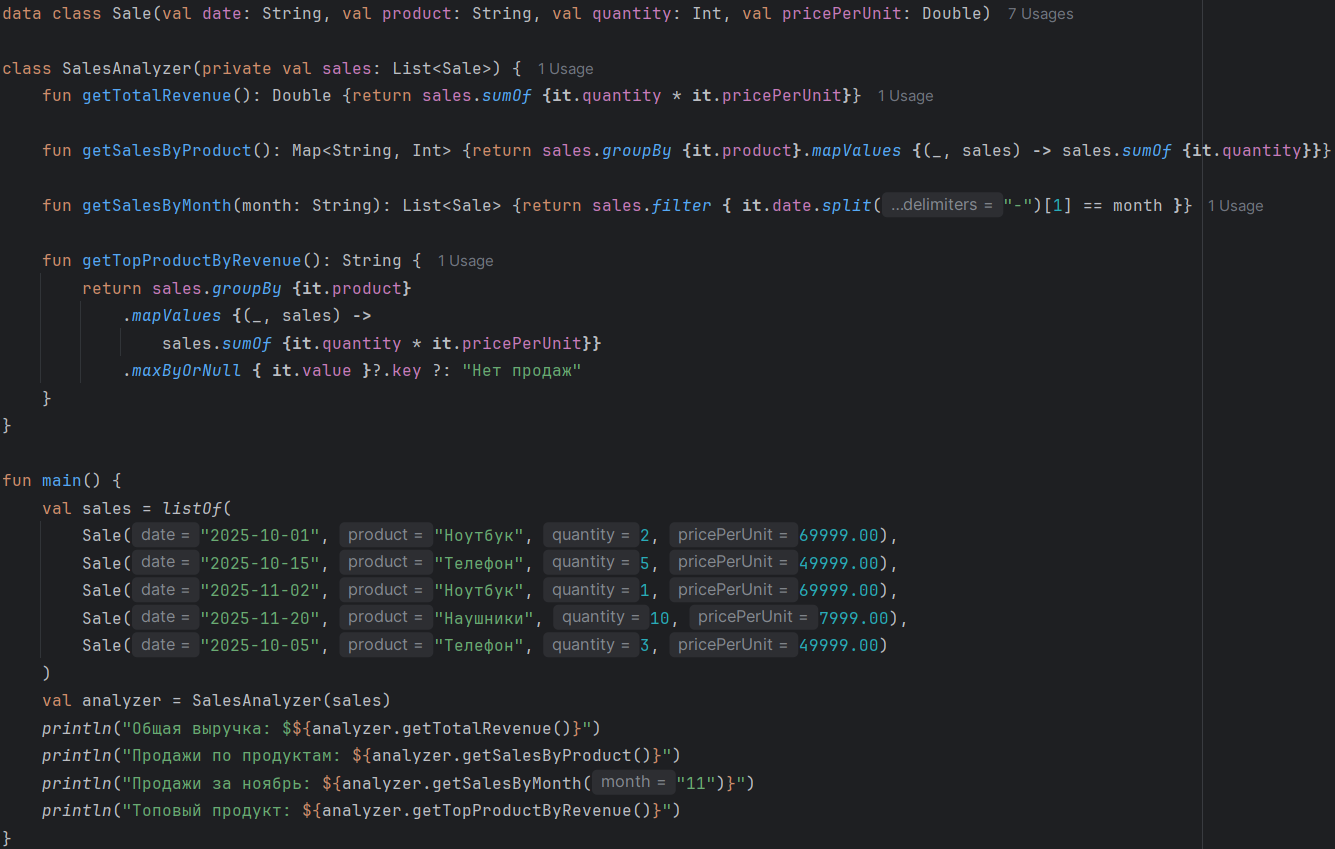


Рисунок 13.3 – Код системы анализа продаж

Результат работы программы представлен на рисунке 13.4

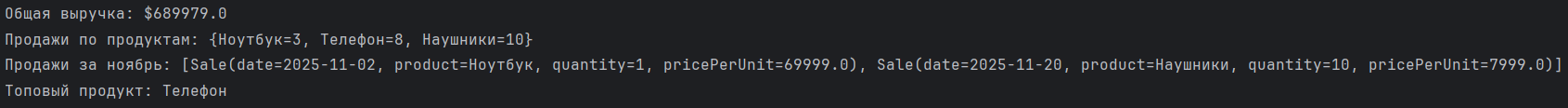


Рисунок 13.4 – Результат выполнения программы для задания 2

## Вывод

В ходе выполнения тринадцатой практической работы были успешно освоены методы работы с коллекциями в Kotlin, включая использование функций высшего порядка для фильтрации, преобразования и агрегации данных, что позволило создать эффективные системы анализа бизнес-данных для интернет-магазина и продаж. Приобретены навыки группировки данных с помощью groupBy(), выполнения сложных агрегаций с sumOf() и mapValues(), фильтрации по различным условиям, а также сортировки и ограничения выборок, продемонстрировав способность решать практические задачи анализа данных с использованием современных средств языка Kotlin.

# Практическая работа 14

## Цель работы

Освоить работу с множествами (Set) и сложными коллекциями (List, Map) в Kotlin, изучить операции над множествами (пересечение, разность, объединение), а также методы фильтрации, группировки и агрегации данных в коллекциях для решения практических задач анализа данных.

## Решение

## Задание 1. Определение пересечения списков

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.

Для решения задачи были использованы два списка чисел, преобразованные в множества с помощью метода toSet(), после чего с использованием операций над множествами были найдены пересечение (intersect) для получения товаров, которые покупались и есть на складе, разность (subtract) для получения товаров, которые покупались, но отсутствуют на складе, и объединение (union) для получения всех уникальных товаров, что позволило эффективно проанализировать соответствие между покупаемыми и доступными товарами.

Код реализации алгоритма программы представлен на рисунке 14.1

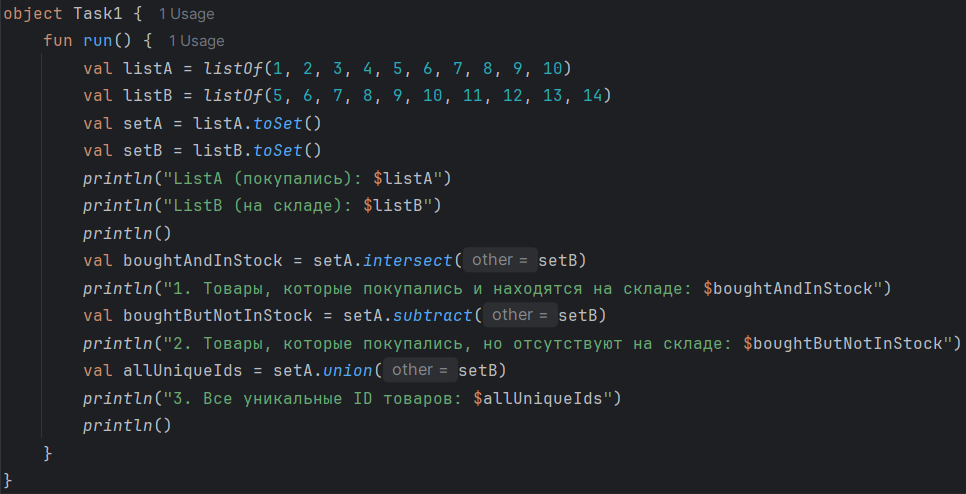


Рисунок 14.1 – Код решения задачи с множествами

Результат работы программы представлен на рисунке 14.2

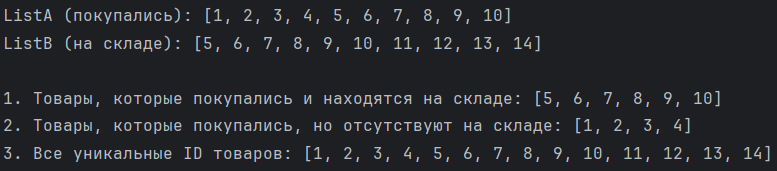


Рисунок 14.2 – Результат выполнения программы для задания 1

## Задание 2. База данных мемов и цитат

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).

1. **Поиск по автору**

Создайте функцию, которая возвращает все записи указанного автора.

1. **Поиск по тегу**

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

1. **Статистику**

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

1. **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.

Для реализации базы данных мемов и цитат была создана неизменяемая коллекция entries, каждый элемент которой представляет собой Map с ключами text, author, type и tags, после чего реализованы функции фильтрации по типу, автору и тегу с использованием методов filter и any, функция получения статистики с извлечением уникальных авторов и тегов через toSet(), а также функция группировки по типу с помощью groupBy, а для взаимодействия с пользователем создано консольное меню с возможностью выбора различных операций и постраничного вывода результатов.

Код реализации алгоритма программы представлен на рисунках 14.3-14.8



Рисунок 14.3 – Код базы данных мемов и цитат (часть 1)

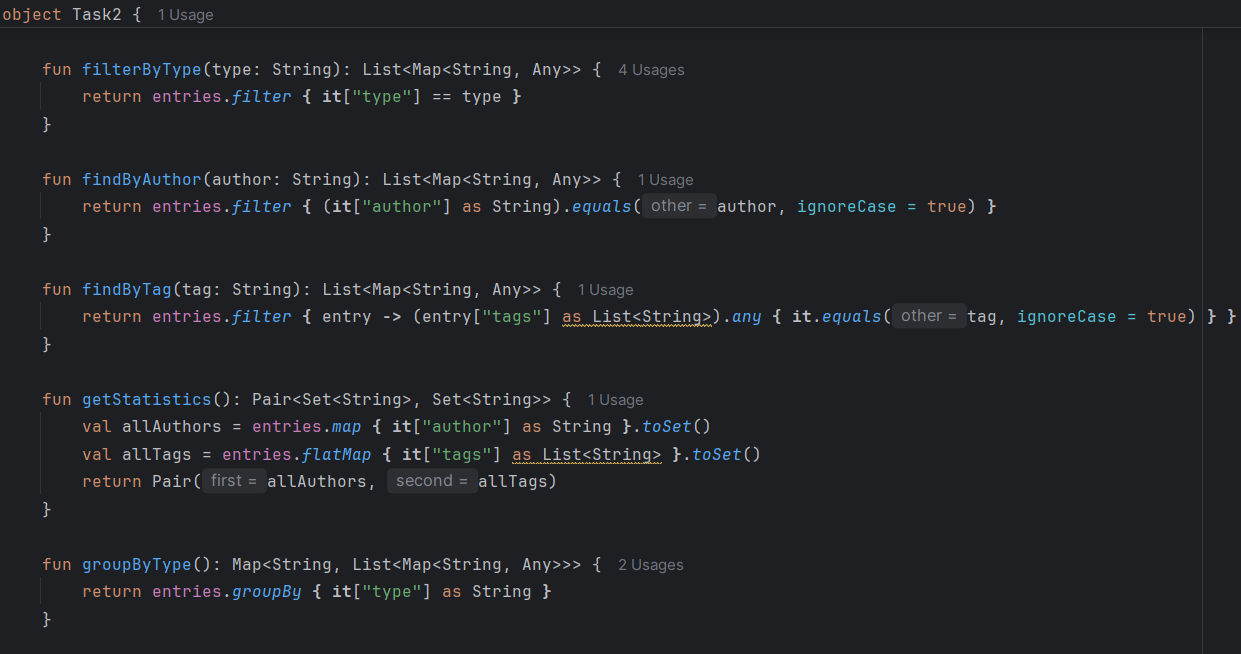


Рисунок 14.4 – Код базы данных мемов и цитат (часть 2)



Рисунок 14.5 – Код базы данных мемов и цитат (часть 3)

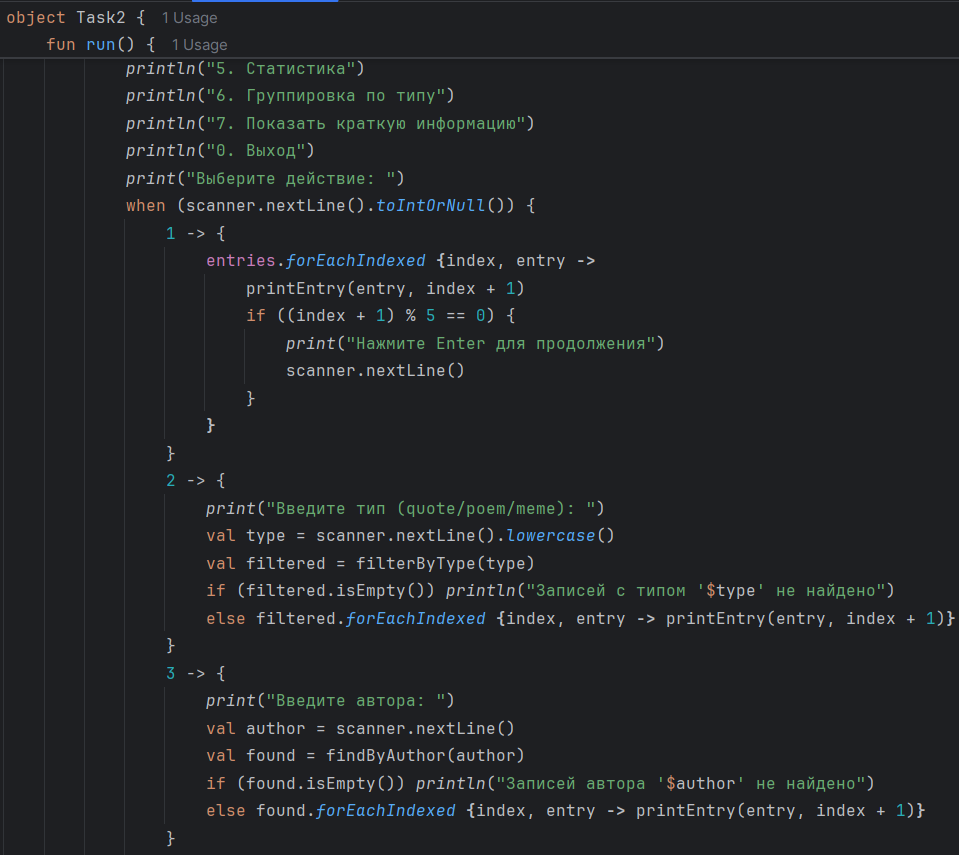


Рисунок 14.6 – Код базы данных мемов и цитат (часть 4)



Рисунок 14.7 – Код базы данных мемов и цитат (часть 5)

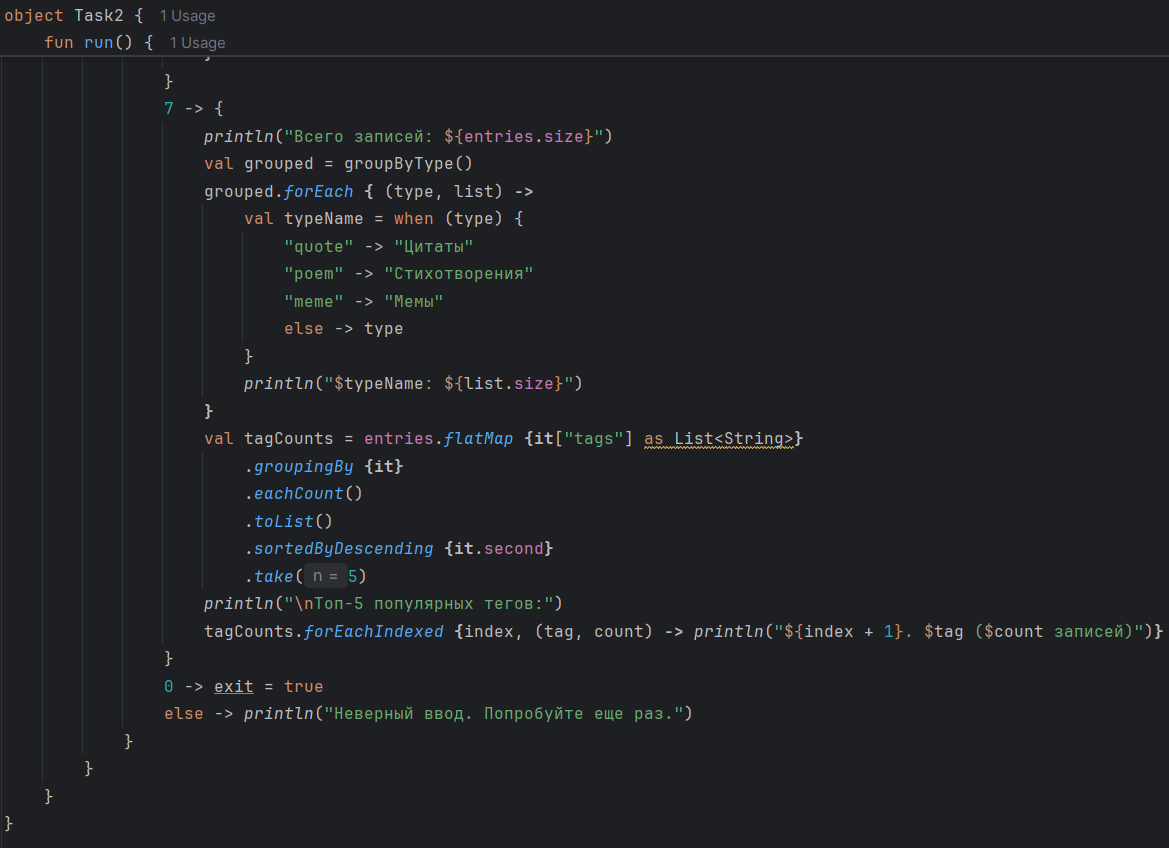


Рисунок 14.8 – Код базы данных мемов и цитат (часть 6)

Результат работы программы представлен на рисунке 14.9

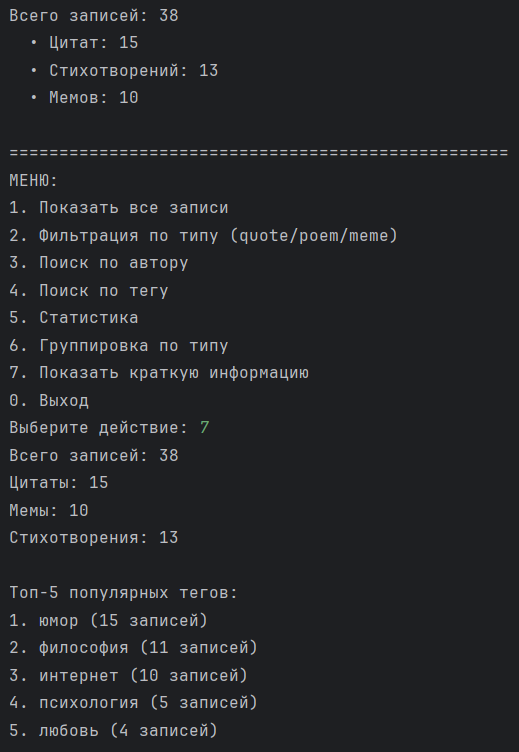


Рисунок 14.9 – Результат выполнения программы для задания 2

## Вывод

В ходе выполнения четырнадцатой практической работы были успешно освоены операции над множествами, такие как пересечение, разность и объединение, что позволило решить задачу анализа товаров, а также применены методы работы со сложными коллекциями, включая фильтрацию, группировку и агрегацию данных, для создания функциональной базы данных мемов и цитат с возможностью поиска и анализа, продемонстрировав эффективность использования встроенных методов Kotlin для обработки данных и создания интерактивных консольных приложений.

# Практическая работа 15

## Цель работы

Освоить работу с последовательностями (Sequence) в Kotlin, изучить методы фильтрации, сортировки, группировки и агрегации данных, а также научиться применять различные функции высшего порядка для обработки коллекций сложных объектов.

## Решение

**Вариант 3. Экологические данные**

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 реализовать:

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.

Создан data-класс CityAirQuality для представления данных о качестве воздуха, после чего реализована последовательность из 15 городов с различными показателями. Для анализа данных разработаны функции, использующие последовательности и функции высшего порядка: filterDangerousCities() для фильтрации городов с опасным уровнем PM2.5, isCountrySafeForPM10() для проверки безопасности всех городов страны по уровню PM10, transformToStatistics() для трансформации данных в текстовый формат, groupCitiesByCountry() для группировки по странам, sortCitiesByPM25() для сортировки по убыванию загрязнения, и calculateCountryAverages() для расчета средних показателей по странам с созданием специального data-класса CountryAverages.

Код реализации алгоритма программы представлен на рисунках 15.1-15.3

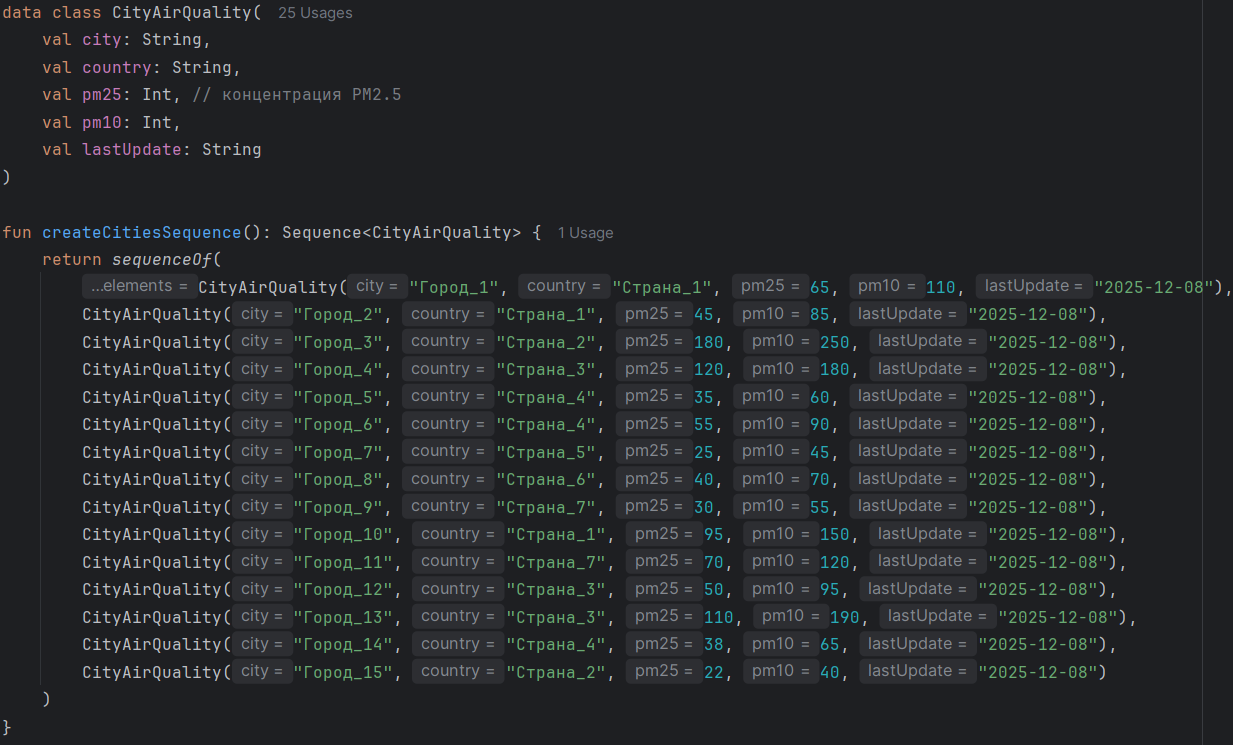


Рисунок 15.1 – Код системы анализа экологических данных (часть 1)

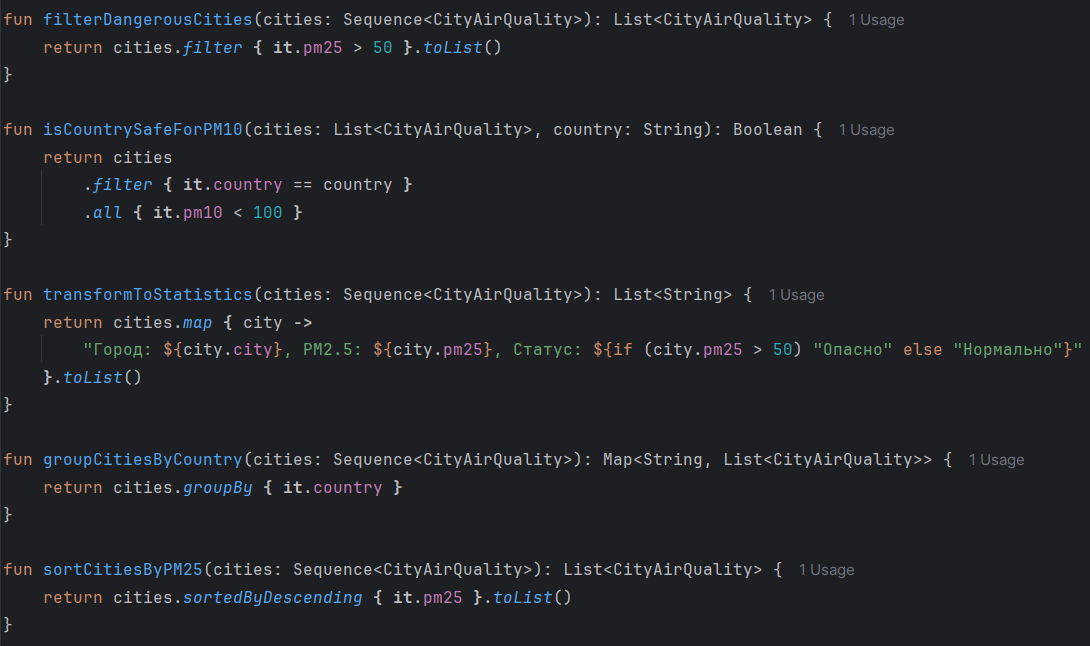


Рисунок 15.2 – Код системы анализа экологических данных (часть 2)

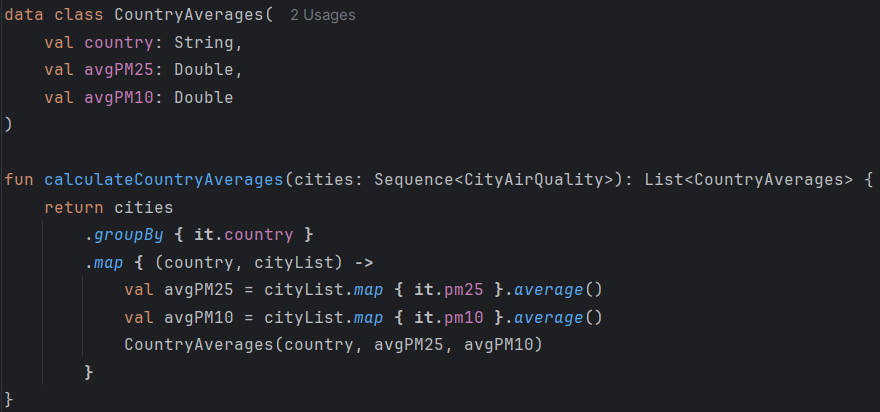


Рисунок 15.3 – Код системы анализа экологических данных (часть 3)

Результат работы программы представлен на рисунках 15.4-15.7

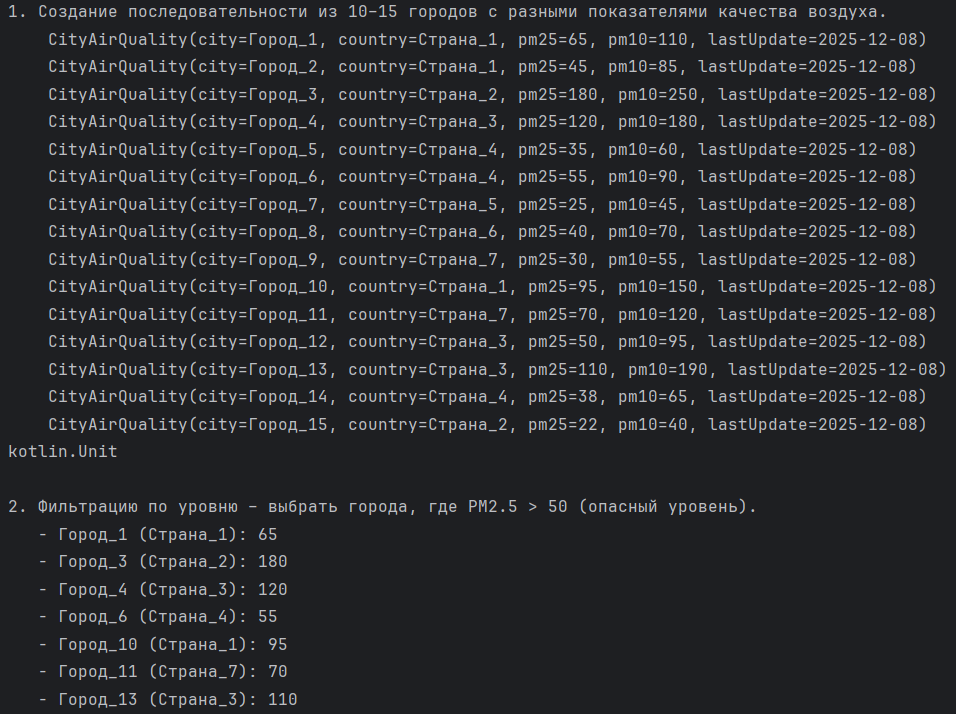


Рисунок 15.4 – Результат выполнения программы (часть 1)

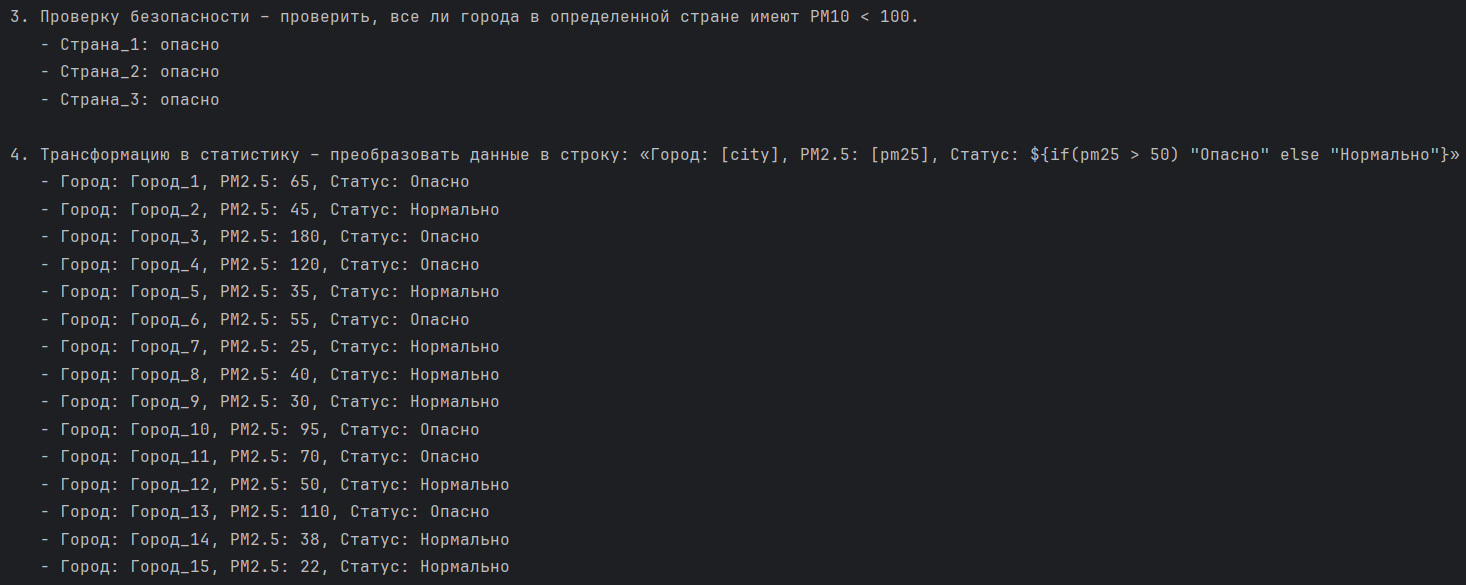


Рисунок 15.5 – Результат выполнения программы (часть 2)

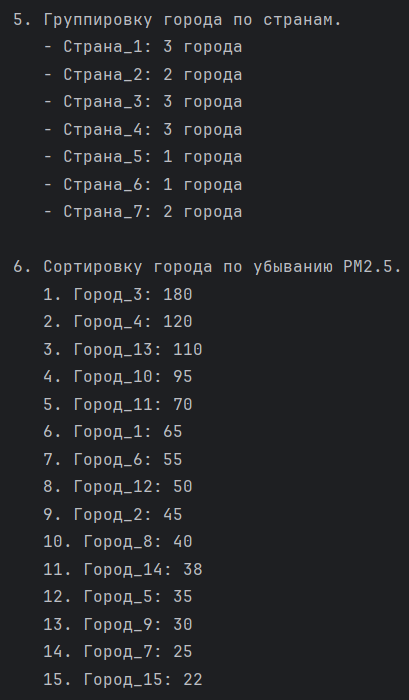


Рисунок 15.6 – Результат выполнения программы (часть 3)

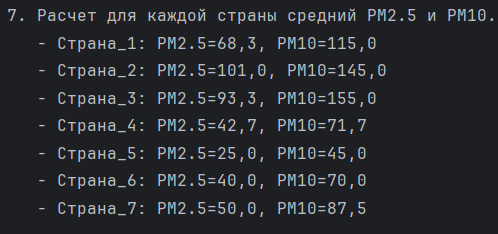


Рисунок 15.7 – Результат выполнения программы (часть 4)

## Вывод

В ходе выполнения практической работы 15 были успешно освоены принципы работы с последовательностями (Sequence) в Kotlin, что позволило эффективно обрабатывать данные о качестве воздуха. Приобретены навыки применения различных функций высшего порядка для фильтрации, сортировки, группировки и агрегации данных, а также преобразования сложных объектов в различные форматы представления. Реализованная система демонстрирует комплексный подход к анализу экологических данных, включая выявление опасных уровней загрязнения, оценку безопасности по регионам, статистическую обработку и ранжирование результатов, что подтверждает готовность к решению практических задач анализа данных с использованием современных средств языка Kotlin.

**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).