Государственное бюджетное образовательное учреждение высшего образования
Московской области
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ
имени дважды Героя Советского Союза, летчика-космонавта А.А. Леонова

Колледж космического машиностроения и технологий

Лабораторная работа

Выполнил:

Студент группы П1-20

Демьянов А. А.

Проверил:

Преподаватель

Стрельников С. Д.

Королёв 2022

Оглавление

1. Описание функций

Все функции программы являются статическими членами соответствующих абстрактных классов в пакете `functions`. Статические свойства и методы являются общими для всего класса, поэтому они могут использоваться без создания экземпляра класса. Арифметические функции находятся в классе `Arithmetic`, прогрессионные и геометрические в классах `Progression` и `Geometric` соответственно.

1.1. Арифметические функции

За данный тип функций в программе отвечает абстрактный класс `Arithmetic` (файл `functions/Arithmetic.kt`). В нём необходимо реализовать 4 статических функции для следующих математических операций:

- Сложение
- Вычитание
- Умножение
- Деление

Реализация функции для сложения двух целых чисел:

```
fun add(a: Int, b: Int) = a + b
```

Листинг #1.1.1. Функция сложения.

Реализация функции для вычитания двух целых чисел:

```
fun subtract(a: Int, b: Int) = a – b
```

Листинг #1.1.2. Функция вычитания.

Реализация функции для умножения двух целых чисел:

```
fun multiply(a: Int, b: Int) = a * b
```

Листинг #1.1.3. Функция умножения.

Реализация функции для деления двух целых чисел:

```
fun divide(a: Int, b: Int) = if (b != 0) a / b else null
```

Листинг #1.1.4. Функция деления.

Таким образом, файл `functions/Arithmetic.kt` примет следующий вид:

Листинг #1.1.5. Класс `Arithmetic`.

```
package functions

abstract class Arithmetic
{
    companion object
    {
        fun add(a: Int, b: Int) = a + b
        fun subtract(a: Int, b: Int) = a - b
        fun multiply(a: Int, b: Int) = a * b
        fun divide(a: Int, b: Int) = if (b != 0) a / b else null
    }
}
```

1.2. Прогрессионные функции

За данный тип функций в программе отвечает абстрактный класс `Progression` (файл `functions/Progression.kt`). В нём необходимо реализовать две статические функции для создания арифметических и геометрических прогрессий.

Реализация функции для создания арифметической прогрессии в виде объекта типа `List<Int>`. В качестве аргументов функция принимает первый элемент прогрессии (аргумент `start`), знаменатель прогрессии (аргумент `step`) и общее количество элементов (аргумент `n`):

Листинг #1.2.1. Функция создания арифметической прогрессии.

```
fun arithmetic(start: Int, step: Int, n: Int): List<Int>
{
    return List(n) { index ->
        start + step * index
    }
}
```

Реализация функции для создания геометрической прогрессии в виде объекта типа `List<Int>`. В качестве аргументов функция принимает первый элемент прогрессии (аргумент `start`), разность прогрессии (аргумент `step`) и общее количество элементов (аргумент `n`):

Листинг #1.2.2. Функция создания геометрической прогрессии.

```
fun geometric(start: Int, step: Int, n: Int): List<Int>
{
     return List(n) { index ->
         start * pow(step, index)
     }
}
```

Таким образом, файл `functions/Progression.kt` примет следующий вид:

Листинг #1.2.3. Класс `Progression`.

```
package functions

abstract class Progression
{
    companion object
    {
        fun arithmetic(start: Int, step: Int, n: Int): List<Int>
        {
            return List(n) { index ->
                start + step * index
            }
        }

        fun geometric(start: Int, step: Int, n: Int): List<Int>
        {
            return List(n) { index ->
                start * pow(step, index)
            }
        }
    }
}
```

1.3. Геометрические функции

За данный тип функций в программе отвечает абстрактный класс `Geometric` (файл `functions/Geometric.kt`). В нём необходимо реализовать три статические функции: первая — для нахождения периметра любых многоугольников (в т. ч. и прямоугольников), вторая и третья — для нахождения площади прямоугольника и окружности.

Реализация функции для нахождения периметра многоугольника. В качестве своего единственного аргумента принимает список длин сторон многоугольника (аргумент `sides`). Результатом работы функции (её возвращаемым значением) является целое число — периметр многоугольника, длины сторон которого находятся в списке `sides`:

Листинг #1.3.1. Функция нахождения периметра.

```
fun perimeter(sides: List<Int>): Int
{
    return if (sides.size == 2)
        2 * (sides[0] + sides[1])
    else
        sides.sum()
}
```

Реализация функции для нахождения площади прямоугольника. В качестве своих аргументов принимает два значения типа Int — ширину и высоту прямоугольника. Возвращаемое значение имеет тип Double:

```
fun rectangleSquare(a: Int, b: Int) = a.toDouble() * b
```

Листинг #1.3.2. Функция нахождения площади прямоугольника.

Реализация функции для нахождения площади круга. В качестве своего единственного параметра принимает длину окружности. Возвращаемое значение имеет тип Double:

```
fun circleSquare(l: Int) = pow(l, 2) / (4 * Math.PI)
```

Листинг #1.3.3. Функция нахождения площади круга.

Таким образом, файл functions/Geometric.kt примет следующий вид:

Листинг #1.3.4. Класс `Geometric`.

```
package functions

abstract class Geometric
{
    companion object
    {
        fun perimeter(sides: List<Int>) = if (sides.size == 2) 2 *
(sides[0] + sides[1]) else sides.sum()

        fun rectangleSquare(a: Int, b: Int) = a.toDouble() * b
        fun circleSquare(l: Int) = pow(l, 2) / (4 * Math.PI)
    }
}
```

1.4. Точка входа в программу

Функция `main` является точкой входа в программу. Она реализована в файле Main.kt (Листинг #5.1 в разделе «Приложение»). Она содержит в себе список регулярных выражений, которые проверяют корректность ввода от пользователя. Затем, через оператор `when`, программа выбирает что необходимо вычислить в зависимости от пользовательского ввода.

1.5. Класс Logger

Запись истории операций и вывод результата реализованы в классе `Logger` (Листинг #5.2 в разделе «Приложение»). Класс содержит в себе три неизменяемые переменные типа `File`, содержащие пути к файлам для хранения истории, а так же две функции для добавления записи в историю (`add`) и получения всей истории (`get`).

2. Описание эксплуатации

При запуске, программа сразу же начинает ожидать строку со стандартного потока ввода (терминал, командная строка). Допустимые варианты для ввода следующие:

1. <Число> <Знак +, -, *, /> <Число>
2. <арифметическая | геометрическая> <Число> <Число> <Число>
3. <периметр> <Не менее двух чисел, разделённых пробелом>
4. <площадь> <Число> [Число]
5. <история> <арифметические | прогрессионные | геометрические>

Случай #1. При вводе двух целых чисел, разделённых знаком «+», «-», «*» или «/» выполняется соответствующая математическая операция. Для «+» — сложение, для «-» — вычитание, для «*» — умножение, для «/» — целочисленное деление. Сами числа могут быть как со знаком «+», так и со знаком «-», или он может отсутствовать (тогда число воспринимается как положительное). Также, между числами и знаком может быть неограниченное число пробелов или они могут отсутствовать вовсе.

Случай #2. Если вводимая строка начинается со слова «арифметическая» или «геометрическая», а за ним три целых числа, то будет создана соответствующая прогрессия. Первое число является первым членом будущей прогрессии, второе число — шаг, последнее — количество элементов. Так же как и случае выше, успешно обрабатываются целые числа как без знака, так и со знаками «+» и «-».

Случай #3. Если вводимая строка начинается со слова «периметр», за которым следует не менее двух целых чисел, разделённых пробелами, то будет высчитываться периметр многоугольника. Если указано минимальное количество чисел (два), то программа вычислит периметр прямоугольника по двум смежным сторонам. Если чисел больше двух, то будет высчитан периметр многоугольника. Допустимы только положительные числа, со знаком «+» или без него.

Случай #4. Если вводимая строка начинается со слова «площадь», за которым следует одно или два числа, разделённых пробелами, то программа высчитает площадь. Если число одно, то оно принимается за длину окружности, а при вводе двух чисел, они принимаются за ширину и высоту прямоугольника. Допустимы только положительные числа, со знаком «+» или без него.

Случай #5. Все вышеперечисленные операции приводили к тому, что их результат выводился в стандартный поток вывода (терминал), но кроме этого, результаты всех вычислений автоматически записывались в файлы. Таким образом, спустя несколько запусков программы, образуется история вычислений, которую можно найти в трёх файлах, размещённых в каталоге «logs». Также, историю вычислений можно запросить для вывода в стандартный поток вывода (терминал, командную строку) у самой программы. Для этого необходимо на ввод подать строку, начинающуюся со слова «история», за которым, на выбор пользователя, можно запросить историю разных типов вычислений: «арифметические», «прогрессионные», «геометрические».

3. Тестирование

Ввод пользователя обрабатывается программой при помощи регулярных выражений. При любом несоответствии им, программа выведет «Недопустимое выражение» в стандартный поток вывода (терминал, командную строку) и завершит свою работу. Каждый тест будет охватывать один тип функций, а последний будет пытаться нарушить работу программы.

Тест #1.

Для проверки арифметических функций будут введены следующие выражения:

Входные данные Теста #1:

```
-3 + 5
10 - 11
9 * 12
28 / 7
```

Вывод соответствует требуемому формату, полученные значения равны ожидаемым:

```
> -3 + 5
-3 + 5 = 2
> 10 - 11
10 - 11 = -1
> 9 * 12
9 * 12 = 108
> 28 / 7
28 / 7 = 4
> выход
```

Рисунок #1. Результаты теста #1.

Тест #2.

Для проверки прогрессионных функций будут введены следующие выражения:

Входные данные теста #2:

```
арифметическая -10 5 6
геометрическая -8 -2 7
```

Вывод соответствует требуемому формату, полученные значения равны ожидаемым:

```
> арифметическая -10 5 6
арифметическая; 5;
[-10, -5, 0, 5, 10, 15]
> геометрическая -8 -2 7
геометрическая; -2;
[-8, 16, -32, 64, -128, 256, -512]
> выход
```

Рисунок #2. Результаты теста #2.

Тест #3.

Для проверки геометрических функций будут введены следующие выражения:

Входные данные теста #3:

```
периметр 3 4
периметр 4 5 3 4
площадь 6 8
площадь 12
```

Вывод соответствует требуемому формату, полученные значения равны ожидаемым:

```
> периметр 3 4
периметр [3, 4] = 14
> периметр 4 5 3 4
периметр [4, 5, 3, 4] = 16
> площадь 6 8
площадь [6, 8] = 48.0
> площадь 12
площадь [12] = 11.459155902616464
> выход
```

Рисунок #3. Результаты теста #3.

Тест #4.

Последний набор входных тестовых данных попытается нарушить работу программы так, чтобы она закончила своё выполнение с ошибкой:

Входные данные теста #4:

```
4 ^ 2
4 / 0
арифметическая 1
площадь 2 5 8
```

При выполнении данного теста, программа не выдала никаких исключений и завершалась удачно с выводом «Недопустимое выражение» или «null» в случае с делением на ноль:

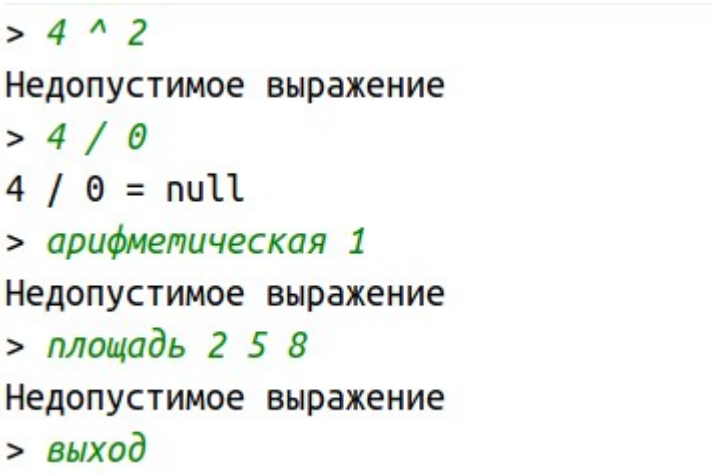

Рисунок #4. Результаты теста #4.

4. Заключение

В ходе выполнения данной работы понадобилось изучить несколько дополнительных тем и особенности работы с ними на языке Kotlin, из которых, в частности, хотелось бы выделить следующие:

- Регулярные выражения
- Абстрактные классы
- Enum-классы
- Статические методы и поля обычных и абстрактных классов
- Работа с файлами (создание, открытие и запись)

5. Приложение

Листинг #5.1. Точка входа в программу.

```
import functions.Arithmetic
import functions.Progression
import functions.Geometric


fun main()
{
    val re = listOf(
        Regex("""^(?<operation>история) (?<arg1>арифметические|
прогрессионные|геометрические)$"""),
        Regex("""^(?<arg1>(?:\+|\-)?\d+) *(?<operation>\+|\-|\*|\/) *(?
<arg2>(?:\+|\-)?\d+)$"""),
        Regex("""^(?<operation>арифметическая|геометрическая) (?<arg1>(?:\
+|\-)?\d+) (?<arg2>(?:\+|\-)?\d+) (?<arg3>(?:\+|\-)?\d+)$"""),
        Regex("""^(?<operation>периметр)(?<arg1>(?: \+?\d+){2,})$"""),
        Regex("""^(?<operation>площадь) (?<arg1>\+?\d+)(?: (?<arg2>\+?\
d+))?$"""),
        Regex("""^(?<operation>выход)$""")
    )

    val logger = Logger("logs", "arithmetic.log", "progression.log",
"geometric.log")

    while (true)
    {
        print("> ")
        val input = readln()

        var regexResult: MatchResult? = null

        for (r in re) {
            regexResult = r.matchEntire(input)
            if (regexResult !== null)
                break
        }
```

```
        if (regexResult === null) {
            println("Недопустимое выражение")
            continue
        }

        val operation = regexResult.groups["operation"]?.value
        val arg1 = try { regexResult.groups["arg1"]?.value } catch (_:
Exception) { null }
        val arg2 = try { regexResult.groups["arg2"]?.value } catch (_:
Exception) { null }
        val arg3 = try { regexResult.groups["arg3"]?.value } catch (_:
Exception) { null }

        when
        {
            operation == "выход" -> break

            operation == "история" && arg1 !== null -> {
                when (arg1) {
                    "арифметические" ->
println(logger.get(LogType.ARITHMETIC).joinToString("\n"))
                    "прогрессионные" ->
println(logger.get(LogType.PROGRESSION).joinToString("\n"))
                    "геометрические" ->
println(logger.get(LogType.GEOMETRIC).joinToString("\n"))
                }
            }

            operation == "+" && arg1 !== null && arg2 !== null -> {
                val a = arg1.toInt()
                val b = arg2.toInt()
                val result = Arithmetic.add(a, b)
                logger.add(LogType.ARITHMETIC, "$a $operation $b =
$result")
            }

            operation == "-" && arg1 !== null && arg2 !== null -> {
                val a = arg1.toInt()
                val b = arg2.toInt()
                val result = Arithmetic.subtract(a, b)
```

```
                logger.add(LogType.ARITHMETIC, "$a $operation $b =
$result")
            }

            operation == "*" && arg1 !== null && arg2 !== null -> {
                val a = arg1.toInt()
                val b = arg2.toInt()
                val result = Arithmetic.multiply(a, b)
                logger.add(LogType.ARITHMETIC, "$a $operation $b =
$result")
            }

            operation == "/" && arg1 !== null && arg2 !== null -> {
                val a = arg1.toInt()
                val b = arg2.toInt()
                val result = Arithmetic.divide(a, b)
                logger.add(LogType.ARITHMETIC, "$a $operation $b =
$result")
            }

            operation == "арифметическая" && arg1 !== null && arg2 !== null
&& arg3 !== null -> {
                val start = arg1.toInt()
                val step = arg2.toInt()
                val n = arg3.toInt()
                val result = Progression.arithmetic(start, step, n)
                logger.add(LogType.PROGRESSION, "$operation; $step;\
n$result")
            }

            operation == "геометрическая" && arg1 !== null && arg2 !== null
&& arg3 !== null -> {
                val start = arg1.toInt()
                val step = arg2.toInt()
                val n = arg3.toInt()
                val result = Progression.geometric(start, step, n)
                logger.add(LogType.PROGRESSION, "$operation; $step;\
n$result")
            }

            operation == "периметр" && arg1 !== null -> {
```

```
                val args = arg1.split(" ").filter { it != "" }.map
{ it.toInt() }
                val result = Geometric.perimeter(args)
                logger.add(LogType.GEOMETRIC, "$operation $args = $result")
            }

            operation == "площадь" ->
            {
                if (arg1 !== null && arg2 !== null)
                {
                    val a = arg1.toInt()
                    val b = arg2.toInt()
                    val result = Geometric.rectangleSquare(a, b)

                    logger.add(LogType.GEOMETRIC, "$operation [$a, $b] =
$result")
                }
                else if (arg1 !== null)
                {
                    val a = arg1.toInt()
                    val result = Geometric.circleSquare(a)

                    logger.add(LogType.GEOMETRIC, "$operation [$a] =
$result")
                }
            }
        }
    }
}
```

Листинг #5.2. Класс Logger

```
import java.io.File


enum class LogType { ARITHMETIC, PROGRESSION, GEOMETRIC }

class Logger(logDirPath: String, arithmeticLogFile: String,
progressionLogFile: String, geometricLogFile: String)
{
    private val arithmeticLog: File
    private val progressionLog: File
    private val geometricLog: File

    init
    {
        val logDir = File(logDirPath)

        if (!logDir.exists())
            logDir.mkdirs()

        arithmeticLog = File("${logDir.path}${File.separator}
$arithmeticLogFile")
        if (!arithmeticLog.exists())
            arithmeticLog.createNewFile()

        progressionLog = File("${logDir.path}${File.separator}
$progressionLogFile")
        if (!progressionLog.exists())
            progressionLog.createNewFile()

        geometricLog = File("${logDir.path}${File.separator}
$geometricLogFile")
        if (!geometricLog.exists())
            geometricLog.createNewFile()
    }

    fun add(lt: LogType, msg: String)
    {
        println(msg)
```

```
        when (lt) {
            LogType.ARITHMETIC -> arithmeticLog
            LogType.PROGRESSION -> progressionLog
            LogType.GEOMETRIC -> geometricLog
        }.appendText("$msg\n")
    }

    fun get(lt: LogType): List<String>
    {
        return when (lt) {
            LogType.ARITHMETIC -> arithmeticLog
            LogType.PROGRESSION -> progressionLog
            LogType.GEOMETRIC -> geometricLog
        }.readLines()
    }
}
```